Fall 2013

Improving Multicore Resource Efficiency and Performance

Syed Ali Raza Jafri
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Syed Ali Jafri

Entitled
Improving Multi-core Resource Efficiency and Performance

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

MITHUNA S. THOTTETHODI, Co-Chair
Chair

T. N. VIJAYKUMAR, Co-Chair

ANTONY L. HOSKING

VIJAY S. PAI

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): MITHUNA S. THOTTETHODI, Co-Chair

Approved by: M. R. Melloch 08/05/2013
Head of the Graduate Program Date

IMPROVING MULTICORE RESOURCE EFFICIENCY AND PERFORMANCE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Syed Ali Raza Jafri

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana

I would like to dedicate this dissertation to my wife for her loving support and encouragement.

ACKNOWLEDGMENTS

I would like to thank Yu-Ju Hong, Gwendolyn Voskuilen, and Hamza Bin Sohail for their help and support.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

Jafri, Syed Ali, Raza. Ph.D., Purdue University, December 2013. Improving Multicore Resource Efficiency. Major Professors: T. N. Vijaykumar and Mithuna Thottethodi.

With clock speeds stagnating for the last few years and multi-cores having replaced uniprocessors, software development must now turn towards shared memory parallel programming to continue enhancing performance. Shared memory parallel programming; however is significantly more challenging than its sequential counterpart. Conventional shared memory parallel programs can fall victim to deadlocks, livelocks and data races which are hard to detect and debug. Aside from programming complexity chip-multiprocessors need a scalable, low latency, high bandwidth interconnect fabric to deliver performance. Conventional interconnects such as crossbars and buses can deliver low latency but do not scale with increasing number of cores. Researchers have proposed the transactional memory (TM) model to address the issue of multi-core programmability and multi-hop on-chip networks to provide low latency, high bandwidth communication among cores. However these designs make inefficient use of resources and also fall victim to performance bottlenecks. TM designs require large amount of memory hierarchy space to store metastate. This design requirement poses a significant barrier to TM adoption by commercial vendors. TM designs also suffer from degraded performance because of current conflict resolution policies. Similarly on-chip networks require a significant fraction of total processor energy, and suffer from performance bottlenecks such as head-of-line blocking and poor switch arbitration. In my dissertation, I make common-case observations to propose novel techniques that considerably reduce resource usage and that significantly improve performance.

1. INTRODUCTION

Multi-cores are emerging as a better alternative to uniprocessors in terms of power dissipation and performance. However, multi-cores pose two key challenges; (1) they require parallel programming, and (2) they require low latency inter-core communication to perform well. The first challenge arises because parallel programming is significantly harder than sequential programming. Conventional shared memory programming models require locking critical sections of threads which can result in undesirable behavior (deadlocks, live locks and data races). The second challenge arises because conventional interconnect fabrics like crossbars and buses cannot scale adequately with increasing number of cores. Researchers have proposed the transactional memory programming model (TM) and scalable mesh network interconnects to address the programmability and inter-core communication issues of multi-cores respectively. However these designs make inefficient use of resources and also fall victim to performance bottlenecks. TM designs require large amount of memory hierarchy space to store metastate. This design requirement poses a significant barrier to TM adoption by commercial vendors. TM designs also suffer from degraded performance because of current conflict resolution policies. Similarly on-chip networks require a significant fraction of total processor energy, and suffer from performance bottlenecks such as head-of-line blocking and poor switch arbitration. In this work, I make common-case observations to propose novel techniques that considerably reduce resource usage and that significantly improve performance.

In Section 1.1, I introduce the resource and performance issues related to TM designs and my proposed mechanisms to address these issues. I then explain how on-chip networks make inefficient use of energy and also degrade performance and my proposed solutions to address these issues in Section 1.2.

1.1. Transactional Memory

Based on databases' transactional processing, transactions achieve atomic behavior without specifying explicit locks by ensuring that the read and write accesses of one transaction do not conflict with another transaction. Transactions can better avoid the above undesirable behavior such as deadlocks, livelocks and data races than locks.

1.1.1. Metastate Overhead

Hardware implementations of transactional memory (HTMs) have made significant progress in providing support for features such as long transactions that spill out of the cache, and context switches, page and thread migration in the middle of transactions. While essential for the adoption of HTMs in real products, supporting these features has resulted in significant state overhead. For instance, TokenTM [7], which is a comprehensive and elegant proposal, adds at least 16 bits per block in the caches which is significant in absolute terms, and steals 16 of 64 (25%) memory ECC bits per block, weakening error protection. Also, the state bits nearly double the tag array size. These significant and practical concerns may impede the adoption of HTMs, squandering the progress achieved by HTMs. The overhead comes from tracking the thread identifier and the transactional read-sharer count at the L1-block granularity. The thread identifier is used to identify the transaction, if only one, to which an L1-evicted block belongs. The read-sharer count is used to identify conflicts involving multiple readers (i.e., write to a block with non-zero count). To reduce this overhead, I leverage the observation that the thread identifiers and read-sharer counts are not needed in a majority of cases. (1) Repeated misses to the same blocks are rare within a transaction (i.e., locality holds). (2) Transactional read-shared blocks that both are evicted from multiple sharers' L1s and are involved in conflicts are rare. Exploiting these observations, I propose a novel HTM, called LiteTM, which completely eliminates the count and identifier and uses software to infer the lost information. Using simulations of the STAMP benchmarks running on 8 cores, I show that LiteTM reduces TokenTM's state overhead by about 87% while performing within 4%, on average, and 10%, in the worst case, of TokenTM.

1.1.2. Performance Issues

Most TMs optimistically allow concurrent transactions, detecting read-write or write-write conflicts. Upon conflicts, existing hardware TMs (HTMs) use one of three conflict-resolution policies: (1) always-abort, (2) always-wait for some conflicting transactions to complete, or (3) always-go past conflicts and resolve acyclic conflicts at commit or abort upon cyclic dependencies. While each policy has advantages, the policies degrade performance under contention by limiting concurrency (always-abort, always-wait) or incurring late aborts due to cyclic dependencies (always-go). Thus, while always-go avoids acyclic aborts, no policy avoids cyclic aborts. I observe that most cyclic dependencies are caused by threads interleaving multiple accesses to a few heavily-read-write-shared delinquent data cache blocks. These accesses occur in code sections called cycle inducer sections (CISTs). Accordingly, I propose the Wait-n-Go (WnG) conflict-resolution policy to avoid many cyclic aborts by predicting and serializing the CISTs. To support the WnG policy, I extend previous HTMs to (1) allow multiple readers and writers, (2) scalably identify dependencies, and (3) detect cyclic dependencies via new mechanisms proposed by Voskuilen et al in [42], namely, conflict transactional state, order-capture, and hardware timestamps, respectively. In 16-core simulations of STAMP, WnGTM achieves average speedups of 46% for higher-contention benchmarks and 28% for all benchmarks over always-abort (TokenTM) with low-contention benchmarks remaining unchanged, compared to always-go (DATM) and always-wait (LogTM-SE), which perform worse than and 6% better than TokenTM, respectively.

1.2. On-Chip Networks

As multi-cores scale in the number of on-chip cores, the superior scalability of multi-hop networks compared to buses and crossbars makes multi-hop networks the choice interconnection strategy. Multi-hop networks are composed of a set of shared router nodes and channels. The channels carry flits from one router node to another. Conventional router nodes use buffers to handle channel contention via backpressured routing. This implies that there are buffers associated with each input port where arriving flits reside until the router can allocate channel and buffer resources for the next hop in

the flits path. Once a flit is assured buffer space on the next hop router it can vie for the crossbar access through the routers switch allocator.

1.2.1. Energy Overheads

A significant part of the networks' energy is consumed in the buffers used to handle link contention via backpressured routing. Recent work proposes to apply well-known backpressureless routing techniques, which eliminate buffers, and hence buffer power (static and dynamic), at the cost of some misrouting/dropping upon link contention (misrouted/dropped flits are eventually recovered/retransmitted). At low loads, misrouting (dropping) is rare and hence backpressureless routing performs well. Unfortunately, backpressureless routers incur significant misrouting/dropping under high loads and saturate at lower throughputs than backpressured networks, resulting in poorer performance and energy. I make the key observation that because load varies significantly across applications, backpressureless and backpressured networks are not robust in performance-energy across the spectrum of high and low loads. That is, at high loads backpressureless networks suffer considerable performance and energy disadvantage compared to backpressured networks; and the energy disadvantage reverses at low loads. To address this robustness issue, I along with my colleagues Yu-Ju Hong, Mithuna Thottethodi and T.N. Vijaykumar propose a novel adaptive flow control (AFC) router which dynamically adapts between backpressured and backpressureless flow control. AFC employs three novel mechanisms, namely local contention thresholds, gossip-induced mode-switch, and lazy VC allocation proposed by Hong et al in [55]. The first mechanism maximizes performance (and minimizes energy) in the common case, and the second mechanism ensures correctness in corner cases. The third mechanism exploits flit-by-flit routing in AFC's backpressured mode to simplify VC allocation and reduces the buffer requirements by a factor of two in AFC's backpressured mode. Simulations using commercial workloads and SPLASH-2 confirm AFC's robustness by showing that AFC achieves performance and energy that are closer to that of the better of backpressured and backpressureless networks.

1.2.2. Performance Issues

Switch allocation has a first-order impact on network performance; and hence on overall system performance. Unfortunately, there is a fundamental tension between quality of switch allocation and clock-speed. On the one hand, sophisticated switch allocation mechanisms such as iSLIP include cross stage dependencies that make it challenging to pipeline. On the other hand, simpler allocation algorithms which are pipelineable (and hence amenable to fast clocks) degrade throughput. I propose a high-performance, adaptive-effort, pipelined switch allocator – apSLIP. apSLIP uses three novel ideas to achieve pipelining of the sophisticated iSLIP allocation algorithm. A key hazard in iSLIP is between grant and request stages. To address this hazard, apSLIP allows superfluous requests to occur and leverage the virtual output queuing architecture which naturally enables easy availing of the corresponding grants. Another key hazard in iSLIP is between the reading and updating of priority counters. To address this hazard, apSLIP uses stale priority values and solve the resulting double-booking problem by separating the arbitration into odd and even streams via privatization of the priority counters. Further, I leverage the observation that iSLIP can exploit multiple iterations to improve its matching strength. However, such additional iterations deepen the pipeline and add to the network latency. The improved matching strength helps high-load scenarios whereas the increased latency hurts low-load cases. Therefore, I propose an adaptive-effort pipelined iSLIP – apSLIP – which adapts between one iteration (shallow pipeline) at low loads and two iterations (deep pipeline) at high loads. apSLIP improves performance by 34% on average on an 8x8 network for a suite of workloads including SPLASH-2 benchmarks and commercial applications.

2. LITETM

2.1. Introduction

Shared memory programming model, which is an easier and a more intuitive programming model for multi-cores, uses locks to protect critical sections within threads. However, locks can lead to correctness problems such as deadlocks, livelocks, and data races. Transactions alleviate such problems by providing atomic behavior without specifying explicit locks. A transactional memory system guarantees atomicity by ensuring that the read and write accesses of one transaction do not conflict with another transaction (i.e., a read from or write to a memory location should not witness a write to the same location from another concurrent transaction). Building on the idea of providing hardware support for TM pioneered in [17], several hardware and software (HTM and STM) and hybrid implementations (e.g., [3,12,14,16,24,26,27,28]) have emerged.

While STMs are slow due to their overhead of software conflict detection for every transactional access (true also for software transactions in HTM-STM hybrids), HTMs use hardware to achieve fast conflict detection. Specifically, by exploiting the fact that both TM and coherence enforce the multiple-reader-single-writer invariant at the block granularity; HTMs optimize performance by piggybacking conflict detection on coherence. That is, HTMs elide conflict detection on cache hits and bundle conflict detection on misses as part of miss processing with little increase in latency. Therefore, I focus on HTMs which now provide support for features such as (1) long transactions that exceed the cache capacity [3,6,7,12,13,24], (2) context switches and page and thread migrations in the middle of a transaction [7,28]; and most recently, (3) avoiding coherence protocol changes, which invariably lead to subtle correctness issues and hinder wide-spread adoption [7]. These features are essential for HTMs to be adopted in real products.

TokenTM [7], a comprehensive and elegant proposal, supports all the above features, but incurs high state overhead. While some HTMs do not incur such overhead (e.g., signature-based HTMs [10,11,27,28]), they do not provide all the above features (as discussed in Section 2.2). Other HTMs which support some of the features using per-block state (e.g., VTM [24], OneTM-concurrent [6]) also incur such overhead. Though I focus on TokenTM, I discuss later that my techniques are applicable to these other HTMs. TokenTM's overhead comes from two sources. First, to allow conflict detection in the presence of L1 cache evictions of transactional blocks, TokenTM maintains a count in the shared L2 of L1-evicted, transactional read sharers. The count can quickly detect conflicts (i.e., if writes encounter a non-zero count). Further, when a block has only one transactional sharer then the storage space for the count can be used to hold the sharer's thread identifier. This identifier serves two purposes (1) to identify the conflicting threads in the case of a conflict, and (2) to allow a transaction access to its own transactional blocks that have either been evicted to lower levels or been moved to other caches by coherence (i.e., to avoid self-conflict which would lead to a livelock). To allow evictions from the L2, the count or the identifier need to be spilled to all the levels of the memory hierarchy, including main memory and even the disk. In addition to the count and identifier, TokenTM employs two state bits per block to distinguish among single reader, single writer and multiple readers. Second, to avoid changes to coherence, TokenTM uses additional bits, the count and identifier in L1, in addition to the traditional R and W bits. Thus, TokenTM incurs significant state overhead (e.g., at least 16 bits per 64-byte block in all levels of the memory hierarchy). In addition, TokenTM requires additional flash-copy support in the L1 for context switches, increasing L1 area and latency.

One may think that the state overhead is a mere 3.3% (16 state bits for 64 data bytes). However, relative overhead does not capture the following two concerns. First, to retrieve the transactional state with the data in memory in one access, TokenTM and other HTMs advocate stealing some of the memory ECC bits to hold the state. (Storing transactional state in regular memory would preserve ECC but require two accesses which would increase bandwidth pressure.) However, stealing as many as 16 bits weakens error protection (e.g., 16 bits correspond to 25% of the 64 SECDED bits per 64-

byte blocks), a concern in soft- and hard-error-prone scaled technologies. Second, an overhead of 16 bits per block in L1 and L2 is significant in absolute terms. For instance, such overhead is equivalent to nearly doubling the tag array in a system with 40-bit physical addresses and 32-KB L1 and 8-MB L2. Because HTMs target commodity multicores where cost is a first-order constraint (as opposed to niche products where high-cost mechanisms may be acceptable), this overhead is a concern. Moreover, as cache and memory size scale in future generations, L1 block sizes are likely to remain around 64 bytes, causing the absolute overhead to grow considerably. These significant and practical concerns may impede the adoption of HTMs in real products, squandering the progress achieved by HTMs.

I propose a novel HTM, called *LiteTM*, to reduce the state overhead of the read-sharer count and thread identifier while supporting all the above features and maintaining high performance. Because the state bits are fundamental to guaranteeing transactional semantics, naively shrinking the state to fewer bits would violate correctness. Any such state reduction needs careful techniques to infer the lost information. LiteTM is based on the key observation that the counts and identifiers are needed neither for conflict detection in all cases nor for identifying conflicting transactions in a majority of cases. Consequently, I completely eliminate the counts and identifiers from the entire memory hierarchy and use software to handle the rest of the cases. LiteTM employs only two state bits per block in L1, L2, and main memory, which are adequate for hardware conflict detection. This overhead corresponds to only 3.3% of the 64 SECDED bits per 64-byte block compared to TokenTM's 25%. Additionally, there is no flash-copying in L1. LiteTM is a new design point in the spectrum of HTMs' hardware-software functionality split. TokenTM uses software to rollback program state upon aborts and to clear transactional state of L1-evicted blocks upon both commits and aborts, while detecting conflicts and identifying conflicting transactions in hardware. In contrast, LiteTM pushes the hardware-software split more towards software and decouples key parts of conflict handling for L1-evicted blocks; conflict detection is still in hardware but the conflicting transactions are identified in software using transactional logs. This decoupling is fundamental and can be applied to reduce the state overhead of other unbounded HTMs

with per block state (e.g., VTM, OneTM-concurrent). Because conflict detection is in hardware for all accesses, LiteTM provides strong atomicity. While LiteTM may seem like another HTM-STM hybrid, there is a key difference: conventional hybrids switch an entire transaction from HTM to STM when even a single transactional block is evicted from the L1, whereas LiteTM uses software *only for the L1-evicted blocks* while continuing to use hardware for L1-resident blocks. Because STMs detect conflicts in software incurring significant overhead for every access, conventional hybrids incur significant overhead as they switch to STM on routine hardware events like evictions. In contrast, LiteTM uses software only for evicted blocks, performing close to HTMs.

Targeting the read-sharer count, I observe that transactional read-shared blocks that both are evicted from multiple sharers' L1s and are involved in conflicts are rare. As mentioned above, the read-sharer count enables fast detection of such conflicts. However, eliminating the count poses a hurdle for clearing L2's and memory's transactional state in the uncommon case of read-shared L1-evicted blocks; without the count, I do not know when the last of the sharers commits or is aborted. To address this issue, I employ a novel *lazy clearing* in software by walking the logs of all the current transactions upon a conflict on an L1-evicted block. Because such all log-walks are expensive, I ensure that this case remains uncommon. LiteTM's two state bits in L2 and memory encode states that isolate the more common cases of single reader or writer for a block, where the state is cleared when the single reader or writer commits or aborts. The lazy clearing in OneTM-concurrent [6] refers to the *lazy update* of the thread identifier without any log-walks and works only in the restricted case where at most one transaction may spill out of L1. In contrast, LiteTM's *lazy clearing of transactional state* handles the general case of multiple, spilled transactions, requiring all log-walks. Furthermore, OneTM's requirement of an identifier per block is not removed by the lazy update.

Targeting the thread identifier, which exists only in single sharer cases (multi-sharer cases have the count), I observe that both uses of the identifier — to identify conflicting transactions and to allow a transaction to access its own blocks — are uncommon. For the first use, most conflicts occur for in-L1-cache blocks where the conflicting transactions are trivially identified by the caches involved in the conflict. For the less-common

conflicts on evicted blocks, I identify the conflicting transactions in software by walking all the transactions' logs (as also done in TokenTM in extremely rare cases). For the second use, I observe that repeated misses to the same blocks are rare within a transaction (i.e., locality holds). For the infrequent case of a transaction accessing its own evicted block, I employ a novel *self log*-walk to check the transaction's own read and write sets.

Finally, targeting TokenTM's *R*, *W*, and additional bits in L1, I observe that I can leverage coherence for implicitly differentiating between reads and writes in the common case, without explicitly using *R* and *W* (not differentiating would lead to many false conflicts, as seen in some STAMP benchmarks). Accordingly, LiteTM uses a single *T* bit and employs the novel idea that conservatively but closely approximates *W* by combining L1 'Modified' coherence state and the T bit. That is, a modified block with the *T* bit set is considered as transactionally written. Because *Modified* and *T* combination is a superset of *W*, this *W*-approximation does not miss any real conflicts. However, false conflicts are possible but only in uncommon cases which I explain later.

The key contributions of this chapter are:

- LiteTM compensates for the loss of information in terms of separate *R* and *W* bits, the read-sharer count, and the thread identifier, respectively, via the following novel ideas: (1) *W* approximation for L1-resident blocks, (2) lazy clearing of transactional state in L2 and memory, (3) self log-walk to identify a transaction's own blocks;
- Using simulations of the STAMP benchmarks running on 8 cores, I show that LiteTM reduces TokenTM's state overhead by about 87% while performing within 4%, on average, and 10%, in the worst case, of TokenTM. LiteTM uses two bits per block in L1, L2, and main memory, whereas TokenTM uses 19 bits in L1, and 16 bits in L2 and memory. In contrast, STMs and HTM-STM hybrids need at least one bit per block in L1, L2, and memory for strong atomicity and an upper bound on hybrids' performance shows at least 44% average performance degradation over TokenTM.

The rest of the chapter is organized as follows. In Section 2.2, I contrast LiteTM to previous proposals. I describe TokenTM in Section 2.3 and LiteTM in Section 2.4. I

describe my experimental methodology in Section 2.5. I discuss my experimental results in Section 2.6 and conclude the chapter in Section 2.7.

2.2. Related Work

Conceptually, TMs maintain metastate in a matrix of memory blocks (rows) and threads (columns) where each entry records read and write accesses for a block-thread pair. The key challenge in making TM support fast is that each access (transactional and non-transactional) requires a lookup of the entire row for the memory block to check for conflict across threads followed by an update of the row with the access, whereas a transaction commit or abort requires clearing of the entire column holding the thread's transactional state. However, quick access to the entire matrix indexed by both rows and columns is hard to implement. To address this issue, HTMs exploit the fact that both TM and coherence enforce the multiple-reader-single-writer invariant at the block granularity to employ the crucial performance optimization of piggybacking conflict detection on coherence (i.e., the functional equivalent of row lookup in my matrix analogy). In addition to optimizing conflict detection, HTMs optimize clearing of transactional state on commits and aborts by flash-clearing the state in the cache whenever all the transactional data fits in the cache (i.e., the column-clear in my matrix analogy). While "coherence+flash-clear" offers a natural way to implement the TM matrix for the case when all transactional state is within the caches, the conceptual 2-D matrix model is impractical when I consider virtualizing TM implementations to accommodate blocks that are evicted and threads that are context switched in/out.

Early solutions maintain spilled transactional metastate in custom hardware or software structures which caused significant hardware complexity [3] or software overhead [12, 24]. More recent TM implementations commonly rely on one of two common simplifications. The first approach uses signature-based TM implementations effectively maintain the entire column (per-thread read/write sets) in hash-based Bloom-filter signatures [10, 11, 27, 28]. They do not maintain any per-block information. Such implementations that omit per-block transactional state are not scalable in (a) system size, since conflict detection requires comparison with signatures of all other threads, which inherently requires broadcast across all hardware thread contexts to effectively lookup the

per address state across all threads, and (b) read/write set sizes, since large transactions cause signature saturation which results in a sharp performance loss when transactions are large [7]. One may think that signature saturation may be eliminated by increasing the size of the signature. However, hardware limitations prevent hash signatures that are large enough to avoid saturation because large signatures slow down *all* accesses [25].

An alternative approach to avoid the signature saturation problem is to associate some state with each memory/cache block (e.g., VTM [24], OneTM-concurrent [6], and TokenTM [7]). As a natural consequence of maintaining per-block state, commits may become slower since the transactional state associated with that transaction (the column in the matrix) that has spilled to memory must be cleared one-at-a-time (unlike cache-bits which may be flash cleared). Although VTM stores per-block metadata, the metadata is not co-located with the corresponding memory block. Instead the metadata is spilled to separate global table (called XADT) that must be searched for conflict detection. In addition to the slow-commit problem, XADT searches slow down conflict detection for all accesses in the presence of any spilled metadata (although some searches may be avoided by using a Bloom filter). OneTM-concurrent overcomes the slow-commit problem by limiting concurrency to at most one overflowed transaction which enables logical clearing of metastate by keeping track of the current overflowed transaction. Metastate of older transactions is implicitly invalid and may be cleared lazily. TokenTM may be viewed as a generalization of OneTM-concurrent that allows multiple “spilled” transactions simultaneously. Unfortunately, both TokenTM and OneTM-concurrent require large amounts of transactional state (16 bits per L1 block). LiteTM’s state reduction techniques are applicable to unbounded HTMs with per-block state (e.g., TokenTM, VTM, OneTM-concurrent).

Finally, hybrid TMs use HTM-based execution as a preferred fast-path and fall-back on a slower STM upon virtualization events (e.g., replacements and context-switch) [14,19]. The limited HTMs in hybrids, though simpler because the HTMs do not need to support virtualization, may add significant hardware complexity (e.g., MetaTM [18] requires coherence changes). Further, hybrid TMs may not achieve strong atomicity (e.g., provide only single global lock isolation [14,18,19]) without additional per-block state

(e.g., using UFO [5]). In contrast, LiteTM offers strong atomicity and outperforms hybrids by a significant margin while requiring modest additional state overhead.

2.3. TokenTM: Background

While my techniques are generic and applicable to other HTMs that use per-block state (e.g., VTM [24], OneTM-concurrent [6]), I choose TokenTM as the base scheme to describe the details of LiteTM because TokenTM comprehensively supports all the features described in Section 2.1. This choice allows us to demonstrate that LiteTM can also support all the features while incurring less state overhead and maintaining high performance.

TokenTM maintains the invariant of multiple readers and single writer for transactional blocks. TokenTM implements the invariant by using an abstraction based on *tokens*, where (1) a transactional read to a block must acquire a token for the block; (2) a transactional write to a block must acquire all the tokens for the block; (3) a transaction commit or abort releases all the tokens acquired during the transaction. Read-write conflicts are detected when a read or a write cannot acquire its requisite number of tokens because a conflicting access already holds some or all of the tokens. TokenTM employs LogTM's transaction log [22] to maintain a transaction's read and write sets, and the previous version of the memory state to rollback memory state upon transaction aborts.

TokenTM addresses two key issues. First, TokenTM allows long transactions that may evict transactional blocks from the cache, by pushing transactional state all the way to memory, as shown in Figure 2.1. While this idea was previously proposed in OneTM which allows only one transaction to spill out of cache, TokenTM generalizes OneTM to allow more than one transaction to spill. Second. TokenTM provides TM support without changing the coherence protocol by using two techniques called token *fusion* and *fission*.

Despite the name, TokenTM does not *require* token coherence. The implementation for TokenTM in [7] assumes a conventional directory-based coherence protocol for private L1s with a shared L2. While every transactional access needs to acquire the appropriate number of tokens, piggybacking on coherence allows cache *hits* "to generate" token(s) locally. Such generation does not lead to undetected conflicts because both coherence and TokenTM enforce multiple-readers-single-writer invariant so that any

conflicting access must trigger a global coherence event. TokenTM piggybacks on this event to detect the tokens held by other transactions and thus the conflict. Any non-conflicting transactional cache miss simply receives the data (via coherence) and the appropriate tokens. All acquires of tokens (reads or writes) are logged. TokenTM maintains a read bit (*R*) and a write bit (*W*) per L1 block to *signify* the holding of one token for a read and all the tokens for a write (first row under TokenTM column in Table 2.1). Transaction commit or abort releases the tokens by clearing the *R* and *W* bits using hardware in some cases and software handlers in the rest. All TM-related software functionality is implemented via escape actions [23].

I provide a high-level summary of TokenTM's state in Table 2.1. However, because TokenTM's state ensures transactional semantics in TokenTM and because my purpose is to reduce the state, a high-level description alone would not suffice. Any detailed description would inevitably involve some subtle correctness issues. The rest of the section gives some of these details.

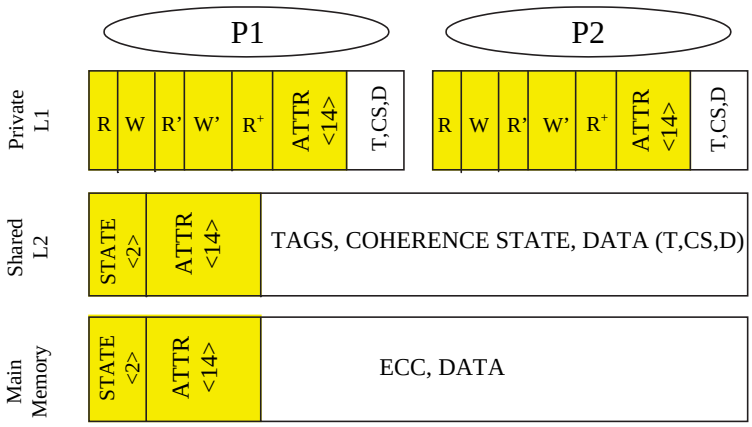

Fig 2.1TokenTM transactional state

2.3.1. Fusion and Fission

It is relatively easy to see that the acquire and release of tokens in non-conflicting cases can be done without changing the coherence protocol. While some previous HTMs,

such as LogTM [22], handle conflicts via nacks which require protocol changes and rollbacks to break deadlocks of mutually-nacking transactions, TokenTM employs fusion of tokens to ensure that the coherence protocol remains unchanged even on conflicting accesses. The no change stipulation requires that coherence actions should *complete as usual even on conflicts* (nacks disallow coherence completion creating potential for deadlocks). A key point here is that though coherence actions complete, the conflicting access does not complete and instead raises an access-fault exception which performs conflict resolution (i.e., rollback of a conflicting transaction). Coherence completion requires that the previously-acquired tokens of the conflicting block must be kept intact through the conflict so that the tokens are released properly either by a commit or abort. I explain how fusion achieves this goal by considering the three cases of conflicts: reads-followed-by-write, write-followed-by-read, and write-followed-by-write.

In a reads-followed-by-write conflict (writer and readers in different cores), the writer invalidates the readers as usual. However, the readers' tokens should not be lost in the invalidated blocks (so the tokens can be released properly in either case of the readers committing or aborting) and the conflict should be detected. To these ends, the readers' tokens are sent to the writer in the invalidation-acknowledgement payloads. Note that adding bits to payloads does not constitute a protocol change as long as there are no changes to states or transitions which are what raise correctness issues. The writer fuses the readers' tokens into its modified block and flags a conflict. Though the readers' tokens are physically present in the writer's block, the tokens belong to the readers and cause the write to fault. The writer proceeds only after the conflict is resolved assuming the writer is not aborted in the resolution. To record the readers' tokens in the writer's block, TokenTM uses the *R'* bit (for single reader), the R^+ bit (for multiple readers), and holds the thread identifier for single sharer or the read-sharer count for multiple readers (Figure 2.1 and second and third rows in Table 2.1) As part of the conflict handling, if the readers are aborted then their tokens are released (i.e., the *R'* in the writer's block is cleared or the read-sharer count is decremented) allowing the writer to acquire all the tokens and proceed. Or if the writer is aborted then there are no tokens to be released because none were acquired (the write did not complete).

In a write-followed-by-read conflict (writer and readers in different cores), the writer's token goes to the reader and sets the W' bit and the thread identifier to indicate a conflict, analogous to the R' bit (second row in Table 2.1). In addition, as part of the modified block write-back, the state bits and thread identifier in the L2 are updated to prevent new reads before the writer commits or aborts. I explain the state bits in the L2 later in Section 3.3. A write-followed-by-write conflict is handled similarly.

While fusion involves many cases as described above, fission is for the relatively-easy case of readers joining non-conflicting sharing where new tokens are "generated" on the fly.

2.3.2. Commits and Aborts

Because fusion occurs only on conflicts without which a transaction's blocks remain in the cache (assuming no evictions which I handle a little later), commit of a transaction that does not encounter any conflicts in its lifetime, is fast. In fast commit, all the acquired tokens are released by a flash-clear of the *R* and *W* bits in the cache. However, the tokens of the transactions, that survive conflicts and reach commit, are fused in other caches. Therefore, such transactions undergo *slow commit* in which a software commit handler performs a log-walk of the read and write sets to release the tokens one at a time. Each token release for a read clears exactly one of an *R* or *R'* bit with matching thread identifier in the absence of which decrements the read-sharer count associated with an R^+ bit (token release uses coherence to contact all the sharers with *R*' and R^+). Each token release for a write clears the *W*, *W'* and L2/directory state bits.

Though the blocks have moved from a transaction's cache, future conflicts on the blocks can be traced back to the transaction through the thread identifiers accompanying *R'* and *W'*, as long as there is only one reader. I call this case as a fast abort. If multiple readers have fused then only the read-sharer count is available and the identity of the readers is lost, requiring future conflicts to walk the logs of all the current transactions to identify the readers. I call this case as a slow abort. Because either cases of abort occur due to conflicts which cause tokens to be fused into nonlocal caches, flash-clear of the *R* and *W* bits is not possible. Consequently, a software abort handler releases the aborted transaction's tokens, similar to the commit handler.

2.3.3. Handling Evictions

One of TokenTM's key features is support for long transactions that spill out of the cache. As L1 blocks get evicted, any tokens they carry are held in the L1 directory (at L2) and in memory when evicted from L2. Each L2 and memory entry maintains two state bits providing four states — *single-reader*, either *idle* (no sharers) or *multiple-reader* (more than one reader), writer, and overflow of read-sharer count — along with the thread identifier/read-sharer count (Fig 2.1 and "Shared L2" rows in Table 2.1). It is easy to see that *R, W, R', W'*, and R^+ can be mapped to these states. The block's thread identifier or read-sharer count is held as is. As more *R, R'*, or R^+ evictions occur (evictions are non-silent), the read sharer count goes up. Accesses from a transaction to its own previously-evicted blocks can be recognized using the identifier on the blocks and proceed without any questions of conflict. Because complete token information is available, conflicts on evicted blocks can be flagged. As in the case of in-cache conflicts, the thread identifier allows a *fast abort* in the *single-reader* case, whereas the *multiple-reader* case requires a *slow abort* (i.e., all log-walk to identify the conflicting transactions).

If a transaction evicts a transactional block then the transaction cannot perform a fast commit, so that a slow commit or an abort ensures that the state in the L2 and/or memory is cleared properly. When the same transactionally-read block (i.e., *R, R'*, and R^+) is evicted and accessed again multiple times, a new token is acquired and the block's read-sharer count increases. Each such new token is logged so it can be released at an abort or commit.

2.3.4. Handling OS interactions

The remaining issues are OS interactions such as page migration, context switch, and thread migration in the middle of a transaction. Because the transactional state can be flushed all the way to memory and even to disk, page migration simply moves the page along with the transactional state. The key issue with context switch is that conflicts between the switched-in and switched-out threads should not go undetected. While evicted tokens or fused tokens (*R', W',* R^+) are accompanied by the identifier or count which detect conflicts, in-cache tokens (*R* and *W*) are not. Therefore, the *R* and *W* bits of

the switched-in and switched-out threads cannot be distinguished. To address this issue, upon a context switch, TokenTM flash-ORs the *R* and *W* bits to the *R'* and *W'* bits, and sets the thread identifier. While *W* and *W'* cannot co-exist due to the implied conflict, TokenTM exploits R^+ to ensure that *R* and *R'* are not both set. Thus, there is no flushing of the switched-out transaction's blocks (i.e., constant-time context switch). Finally, thread migration (preceded by a context switch) can occur without any problems. The blocks that are already accessed by the migrated thread are identified by the thread identifier which is preserved through the context switch, and access to other blocks requires new tokens as usual.

Table 2.1TokenTM vs. LiteTM : Transactional state for conflict detection

	TokenTM		**LiteTM**	
	State	Interpretation	State	Interpretation
Private L1	*R/W*	Local thread has transactionally read/written to block	*T+Clean,/ T+Modified*	Local thread has transactionally read/written to block
	R'/W' with ID	Remote transaction ID has read/written to block	*T'*	At least one unknown transaction has accessed the block
	R^+,count(=N)	At least N remote transactional readers exist	--	No explicit tracking of multiple remote transactional accesses
Shared L2, memory (L1-evicted transactional states. Four States in L2	*Single Reader (ID)*	Transaction (ID) has read the block	*Single reader*	Some transaction has read block
	Multiple readers with count N	A total of N transactions (identities unknown have read the block)	*Multiple-reader*	Multiple readers (numbers and identities unknown) have read block
	Single Writer (ID)	Transaction (ID) has written to block	*Single-writer*	Some transaction has written to block
	Idle (encoded as multiple reader with N =0)	No transactional state from blocks has overflowed from upper caches.	*Idle*	Same as TokenTM's state

2.3.5. State Overhead

While comprehensive in supporting all the desired features, TokenTM incurs state overhead at all the levels of the memory hierarchy (19 bits per block in L1, and 16 bits in L2 and memory). As mentioned in Section 2.1, TokenTM and other TMs [5] advocate stealing some of the ECC bits in memory to hold the transactional state which can then be retrieved in one access with the data. The idea is that while SECDED for 64 bits requires 8 bits, SECDED for 256 bits requires only 10 bits, thus sparing 22 bits for SECDED-protected transactional state. However, stealing as many as 16 bits weakens error protection (e.g., 16 bits are 25% of the 64 SECDED bits per 64-byte blocks). Also, 16 bits per block is a significant overhead in absolute terms, equivalent to nearly doubling the tag array in a system with 40-bit physical addresses and 32-KB L1 and 8-MB L2.

Table 2.2TokenTM vs LiteTM

	TokenTM vs LiteTM	Missing information	**LiteTM's compensation**	Performance impact on LiteTM
Private L1	*R,W,R',W',R+*, and 14 bits of count/id(19 bits) vs. *T,T'*	R and W not separate	Approximate *W* by '*Modified*' and *T* (in hardware)	Extra false conflicts – extra fast-aborts (conflict on L1-resident block); slow-aborts (conflict on evicted block)
		No thread id	Self/all log-walk for potential conflict – hit or miss to *T'* (in software)	Self log-walk overhead(no conflict) – slow commit in both TMs; all log-walk overhead(conflict) – fast-aborts in TokenTM becomes slow-aborts
		No read-sharer count	Abort all but one reader for multiple-reader conflict (in software)	Extra aborts – extra fast-aborts
Shared L2 and memory	2 state bits and 14 bits of count/id vs. 2 state bits	No thread id	Self/all log-walk for potential conflict on evicted single-reader or writer block (in software)	Self log-walk overhead (no conflict) – slow-commit in both TMs; all log-walk overhead (conflict) – fast-aborts in TokenTM becomes slow-aborts
		No read-sharer count	Lazy clearing all log-walk for a write's potential conflict on evicted multiple – reader block (in software)	All log-walk overhead – fast – or slow-commit depending on evictions in both TMs (no conflict); slow-aborts in both TMs (conflict)

2.4. LiteTM

Recall from Section 2.1 that I propose LiteTM to reduce TokenTM's state overhead based on the key observation that read sharer counts and thread identifiers are not needed for conflict detection. Even to identify conflicting transactions, the state is not needed in a majority of cases (i.e., when the transactional state has not been evicted from L1). As such, I completely eliminate the counts and identifiers from the entire memory hierarchy. LiteTM decouples key parts of conflict handling for L1-evicted blocks; conflict detection is still in hardware but the conflicting transactions are identified in software using transactional logs. LiteTM employs only two state bits per block in L1, L2, and main memory, which are adequate for conflict detection in hardware. Because conflict detection is in hardware for all accesses, LiteTM provides strong atomicity.

To eliminate the read-sharer count, I observe that transactional read-shared blocks that both are evicted from multiple sharers' L1s and are involved in conflicts, which are detected by the count, are rare. To eliminate the thread identifier, which exists only in single-sharer cases (multi-sharer cases, instead, use the count), I observe that both uses of the identifier — to identify conflicting transactions involved in a conflict on an L1-evicted block and to allow a transaction to access its own blocks — are uncommon. The first use is uncommon because most conflicts occur for in-L1- cache blocks where the conflicting transactions are trivially identified (by the caches involved in the conflict). The second use is uncommon because repeated misses to the same block are rare within a transaction (i.e., locality holds). Finally, I replace TokenTM's *R, W, R', W'*, and R^+ bits in the L1 with only *T* and *T'* bits by observing that I can leverage coherence for implicitly differentiating between reads and writes in the common case, without explicitly using *R* and *W*. I employ the novel idea that conservatively but closely approximates the *W* bit by combining L1 *'Modified'* state and the *T* bit.

In the rest of this section, I explain how LiteTM uses software to handle the uncommon cases, for which TokenTM uses separate read and write bits, the count, and the identifier. Table 2.2 shows a high-level summary of these differences. As mentioned in Section 2.1, because TokenTM's state bits are fundamental to guaranteeing

transactional semantics, naively shrinking TokenTM's state to fewer bits would violate correctness. LiteTM carefully compensates for the lost information.

2.4.1. Modifications to transactional state bits: T (transactional) bit in L1 and two bits in L2, memory

While TokenTM uses *R* and *W* bits in L1, LiteTM merges read and write into a single T bit (transactional bit) (first row under LiteTM column in Table 2.1). As in TokenTM, a transactional read or write hit locally sets the T bit. If multiple readers concurrently get cache hits, then multiple T bits are set, as are multiple *R* bits in TokenTM. To detect conflicts, I need to infer that a given block was transactionally written using a single *T* bit and no *W* bit. To that end, I approximate *W* by considering modified blocks with the *T* bit set to be transactionally written. Thus, any request to the modified block gets a reply with the modified state and the *T* bit as part of the payload, allowing the requestor to detect the conflict and incur a fault.

Because *Modified* and *T* combination is a superset of *W*, the *W* approximation does not miss any real conflicts. However, false conflicts are possible but only in the following case: A block is non-transactionally modified, or transactionally modified and committed. Then a new transaction on the same core reads the block which becomes modified and transactional (i.e., approximated as a transactional write). Finally, a remote transaction reads the block, resulting in the abort. However, if the remote read occurs *before* the local read then there is no abort because the block would not be transactional at the time of the remote read. With even three or more read sharers, the chances of the local read occurring first and causing the false abort are low. Also, if there is no read sharing then there are no false aborts. Therefore, such false aborts are rare in general. I could have avoided the false aborts by confirming the conflict via a log-walk of the writer transaction. However, such log-walks are pure overhead for true conflicts, which are more common than false conflicts. Therefore, I do not perform this log-walk (first row in Table 2.2). Note that *W*-approximation does not impact cache hits in any way (i.e., transactional write hits to modified blocks with or without *T* bit set proceed as in TokenTM).

W-approximation is recorded in the transactional state, called writer-state, in the L2 (and memory) whenever a modified L1 block with the *T* bit set is evicted or is fused upon a conflict. LiteTM's writer-state is similar to that of TokenTM's though TokenTM's state is exact. I explain fusion details next and eviction details in Section 2.4.4.

2.4.2. Modification to Fusion: T' and log-walks

To avoid coherence changes to handle conflicts, LiteTM employs fusion but with some modifications. First, reads-followed-by-a-write conflicts in LiteTM (writer and readers on different cores) fuse the readers' tokens at the writer, as in TokenTM. However, LiteTM does not have read-sharer counts in the L1 and there is only a single *T'* bit to track exactly one reader's token (second and third rows in Table 2.1). Consequently, all but one reader are always aborted in this type of conflict so that either the writer or exactly one reader survives (third row in Table 2.2). Because conflicts involving multiple read-sharers are not common, LiteTM's extra aborts over TokenTM do not degrade LiteTM's performance by much.

Second, in write-followed-by-read conflicts, the usual modified-block writeback is accompanied with the T bit in the payload, allowing the transactional state in the L2 (and memory) to go to the *writer-state*. Just as TokenTM sets the *W'* bit in the reader's block, LiteTM sets the *T'* bit (second row in Table 2.1). Accesses or miss requests to blocks with the *T'* bit set can neither differentiate whether the *T'* bit is from a read or a write, nor identify the transaction whose *T* bit was converted into the *T'* bit given that LiteTM does not have thread identifiers. Consequently, such an access raises a potential-conflict exception so that the exception handler performs log-walks of all the current transactions to determine whether there is a conflict. I discuss evictions of *T'* blocks in Section 2.4.4.

To avoid unnecessary all log-walks when the block is already in the accessor's read or write set (as appropriate), the accessor first performs a lower-overhead *self log-walk* of its own log and triggers an all log-walk only if the block is not in the accessor's read or write set (second row in Table 2.2). The log-walks are optimized to look up only the read set or the write set where appropriate (e.g., only the write set for *self log-walk* by a write) and to scan the log starting from the end to find the block sooner due to locality.

Fortunately, such all log-walks are not frequent as they correspond to read-shared access to a previously-conflicted block, and hence do not degrade performance much.

Finally, there are some subtle details about log-walks. While one thread performs an all log-walk due to a conflict, other threads can continue concurrently and update their logs. Because all conflicts result in a fault and perform a retry, there is no risk of permanently missing a conflict. A new access, N, that conflicts with the faulting access, F, and intervenes or races with F's log walk may be missed temporarily by the log walk. However, N would take coherence permissions away from F which upon retry, would coherence miss, fault again, do a log walk, and catch the missed conflict. To avoid unlikely, indefinitely-repeated retries, I stop all other threads after a fixed threshold on the number of retries (this condition did not occur in my runs).

2.4.3. Modification to Commits and Aborts: Log-walks

As in TokenTM, transactions that do not encounter any conflicts (hence, their tokens have not moved) undergo fast commits in LiteTM, whereas transactions that survive conflicts undergo slow commits. In the case of aborts, because there is no thread identifier in LiteTM, only in-cache conflicts on *T* blocks can identify the conflicting transactions and undergo fast aborts (i.e., no all log-walks). All conflicts on evicted blocks require all log-walks, as do in-cache conflicts on *T'* blocks, as discussed before.

2.4.4. Modifications to handling evictions: Lazy clearing

One key difference from TokenTM pertains to token release for L1-evicted blocks. Tokens in L1-evicted blocks are fused into the L2 which may spill into memory, as done in TokenTM. LiteTM's L2 and memory have two state bits per block to record the following states which are similar to TokenTM's states (Section 3.3) as well as the directory states in [4]: idle, single-reader, writer, and multiple-readers ("Shared L2" rows in Table 2.1). A clean, *T* or *T'* block that is evicted starts in the single-reader state in L2 and goes to the multiple-reader state if more such copies are evicted. If a modified, *T* block is evicted; the block enters the writer-state in L2. A modified, *T'* block cannot exist due to the implied conflict.

Because there is no identifier, a transaction cannot recognize its own evicted block in the single-reader or writer states and must perform a *self log-walk*. As discussed in Section 4.2, if the *self log-walk* determines that the appropriate token is not already held, then an all log-walk follows to identify the conflicting transactions (LiteTM in second last row in Table 2.2). Because transactions do not miss repeatedly on their own blocks, these log-walks are uncommon.

TokenTM combines *idle* and *multiple readers* cases into one state and uses the read-sharer count to separate the cases. In contrast, because LiteTM does not have the count, LiteTM's *idle* and *multiple-readers* must be separate states. Because *single-reader* and *writer-state* imply only one sharer that has evicted the corresponding block, any commit or abort that tries to clear these states must be from that sharer and hence can proceed. In the case of *multiple-reader* state, however, only the last sharer's commit or abort can clear the states. But because there is no count in LiteTM, the last sharer cannot be distinguished from the rest. Consequently, all clearings of the *multiple-reader* state are ignored by the hardware, causing the read sharers to leave behind this state.

When a conflicting access occurs due to this left-behind state, LiteTM employs *lazy clearing* which performs an all log-walk to determine whether the previous sharers still exist. If so, there is a true conflict requiring an abort, and if not, the state is cleared allowing the access to proceed (last row in Table 2.2). Note that if there is a non-conflicting access (i.e., a read) to the left-behind multiple-reader state, the access can proceed without any lazy clearing or log-walks. Fortunately, conflicts with multiple L1-evicted readers, and hence lazy clearings, are uncommon.

There are two correctness issues: a minor one with *self log-walks* and a major one with lazy clearing. The issue with *self log-walks* involves the retry semantics of faults. *Self log-walks* may evict the block for which the log-walk was triggered. Such evictions would cause another *self log-walk* upon retry, potentially resulting in a livelock. I resolve this issue by touching the block at the end of the *self log-walk* to bring the block into L1. In rare cases, a concurrent conflicting access may steal the block's coherence permissions away between the touch and retry, in which case the retry would fault. Repeated

occurrences of such stealing are possible, though extremely rare, and would be caught by my retry threshold.

The major correctness issue with lazy clearing involves a race. It is possible that after the all log-walk checks a transaction's log and does not find the block in the read set, but before the state is cleared, the transaction may read miss on the block and proceed with the read as the state is multiple-reader. Then, the state clearing would be incorrect. This issue does not arise in TokenTM because TokenTM's log-walks decrement the read-sharer count while new readers increment the count. Increments and decrements are commutative, unlike setting and clearing, so that a reader's increment can come before or after a log-walk's decrement without making the count incorrect.

One option is to stop all other threads during such a lazy clearing but this option would be slow. Another option is to use an extra state bit per block in L2 and memory so that a lazy clearing starts by changing to busy state. Access to a busy block triggers a potential-conflict exception whose service is serialized after the lazy clearing. Because this option increases the state overhead from 2 bits to 3 bits per block, I explore a third option by observing that only a few blocks undergo lazy-clearing at any given moment (e.g., 3-4). Therefore, I employ a few buffers, called busy buffers, in the L2 and memory controller to hold the blocks' addresses. The buffers are common to all the threads of a process. Lazy clearing starts by placing the block address in the buffer and removing the address upon exit. Misses that address-match on a buffer trigger potential-conflict exceptions which are serialized after the lazy clearing in software (without any hardware stalling). For the rare case of exceeding the number of buffers, I employ a single counter for a process, called busy counter, to track the excess lazy clearings. Any miss that encounters a non-zero busy counter triggers a potential-conflict exception irrespective of address-match on a buffer, and is serialized after all the current lazy clearings.

2.4.5. Modifications to handling OS interactions

Any context switch in the middle of a lazy clearing, though rare, must preserve the busy buffers and busy counter for correct operation upon resumption. Thus, the buffers and counter are part of the process state. To reduce the amount of the process state, only the sum of the busy counter and the number of non-empty busy buffers is saved. Upon

resumption, the sum is loaded into the busy counter. Thus, in this rare case, all misses trigger potential-conflict exception until all the resumed lazy clearings are complete and the busy counter goes to zero. Note that because of the replay semantics of conflicting accesses as discussed in Section 2.4.2, there is no risk of missed conflicts while a transaction is switched out.

LiteTM handles mid-transaction page migration similar to TokenTM but deals with mid-transaction thread migration differently. While TokenTM leverages R^+ to guarantee that *R* and *R'* are not both set (Section 3.4), LiteTM does have an R^+ -equivalent to give the same guarantee for *T* and *T'*. Therefore, the switched-out thread performs a *self log-walk* in LiteTM and flushes the transactional blocks to memory, so that conflicts with the switched-in thread are detected correctly. Because context switches are rare, such flushing being slow may not be a concern.

2.4.6. Multithreaded hardware support

LiteTM can support multithreaded cores by replicating the *T* bits per hardware thread context while continuing to keep a single *T'* bit per block. Each transaction can infer the existence of other transactional accesses via *T'* (remote) or *T* (another local context). In TokenTM's case, though it has thread identifiers, the identifiers are used for tracking transactional accesses solely from remote cores and not from local contexts. As such, even TokenTM has to replicate its *R/W* bits for each context. As in LiteTM, TokenTM need not replicate *R', W'*, and the attribute bits for each context

2.4.7. LiteTM's generality

LiteTM decouples key parts of conflict handling for L1-evicted blocks; conflict detection is in hardware but the conflicting transactions are identified in software using transactional logs. This decoupling is fundamental and can be applied to other unbounded HTMs using per-block state. For instance, LiteTM can eliminate OneTM-concurrent's thread identifiers [6]), and VTM's identifiers (i.e., pointers to XSW in XADT) and implicit counts (i.e., number of entries in XADT) [24].

2.4.8. State Overhead

LiteTM needs only two state bits per block in L1, L2, and main memory, while TokenTM needs at least 16 bits. Assuming that the ECC memory bits are stolen to hold transactional state, this overhead corresponds to only 3.3% of the 64 SECDED bits per 64-byte block compared to TokenTM's 25%. Also, LiteTM's two bits add about 12% to the tag entries compared to TokenTM's overhead of nearly 100%.

To reduce the state overhead even beyond LiteTM, I experimented with a LiteTM variant that has one state bit per L1 block in L2 and memory. Because the bit cannot distinguish between transactional reads and writes, and single and multiple readers, this variant employs self and all log-walks to make these distinctions. The bit is lazy-cleared, if possible, upon a potential conflict. I show this variant's performance in Section 6.1. Another variant is to have *R, W, R', W'* in L1 (and two state bits per block in L2 and memory) which would reduce L1 overhead from 19 bits to 4 bits, as compared to TokenTM. This variant may be acceptable because L1 is a custom structure unlike main memory. However, because *W*-approximation works well in practice, this variant does not offer any major advantages over the original LiteTM. I emphasize that the bulk of the state reduction comes from removing the thread identifier and sharer count, and not from collapsing *R* and *W* into T.

Table 2.3Hardware parameters

Processors	8, 1 GHz, in-order issue
Private L1	32K, 4-way, 64-byte blocks, 1-cycle latency
Private L2	8M, 8-way, 64 byte blocks, 34-cycle latency
Memory	8GB, 448-cycle latency
Coherence	Directory MOESI with full bit-vector sharer list
TokenTM	n statebits + 14 bits of thread id/sharer count per block in L1 (n = 5) and in L2 and memory (n = 2)
LiteTM	2 state bits per L1 block in L1, L2, memory+4 busy buffers + 1 busy counter

2.5. Methodology

To evaluate my ideas, I implement LiteTM in the Wisconsin GEMS HTM simulator [21] which uses Simics [20] to perform full-system simulations. I simulate a SPARC-based multicore running Solaris 10. Table 2.3 summarizes the parameters of the simulated system. Using GEMS's user-level exception handlers, I faithfully capture all the cases requiring self log-walks, all log-walks, lazy clearing, slow-commits, fast-aborts, and slow-aborts. I use STAMP with the smallest input dataset [9] (many benchmarks do not scale beyond 4 cores with larger datasets, which also slow down simulations). Table 2.4 characterizes the benchmarks showing the fraction of TokenTM's execution time spent in transactions (%xact time), the length of transactions expressed quantitatively as the number of instructions per transaction (#instrs/xact) and qualitatively (xact length), and the amount of contention expressed quantitatively as the ratio of number of aborts to number of commits in TokenTM (#aborts/#commits) and qualitatively (contention). These data mostly match the STAMP paper [9] and show that STAMP covers a wide spectrum of transactional behavior (i.e., evictions, read-sharing instances, and conflicts), even for the small dataset, providing confidence in the generality of my results. Because I compare LiteTM against TokenTM, I validate TokenTM's performance obtained by us against the TokenTM paper [7]. In the column TokenTM vs. LogTM-SE, I show TokenTM's performance normalized to that of LogTM-SE with 2K-bit Bloom filters (2Hx3 in [7]). On average, TokenTM performs 42% better than LogTM-SE (not shown). The numbers agree with the TokenTM paper [7] and show that Bloom filter saturation (Section 2.2) hurts performance, justifying TokenTM's "full-map" approach of per-block transactional state without hashing. In the last column (TokenTM vs. Single), I show TokenTM's speedups on 8 cores over single-thread runs to confirm that TokenTM scales well over at least a modest number of cores. The benchmarks with long transactions and high contention (e.g., yada, bayes, and labyrinth) perform better if the oldest conflicting transaction is chosen to survive the abort, as proposed in [8] to alleviate the problem of "the starving elder". For uniformity, I apply this policy to all the benchmarks. In LiteTM, I apply this policy also to abort of all but one L1-resident read-sharer (Section 4.2). To account for statistical variations (e.g., the randomized back-off delay for a transaction

relaunch after an abort for reducing repeated conflicts [8]), I use enough randomly-perturbed runs to achieve 95% confidence [2].

2.6. Experimental Results

LiteTM eliminates TokenTM's read-sharer count and the thread identifier to reduce the state overhead from 16 bits per L1 block to 2 state bits. To compensate for this missing information, LiteTM perform parts of conflict detection in software for L1-evicted blocks. That is, conflict detection is still in hardware using the state bits but the conflicting transactions are identified in software by walking transactional logs. In doing so, LiteTM incurs some performance overhead. Therefore, I begin with a performance comparison of LiteTM against TokenTM. I then explain the performance numbers by quantifying the number of self and all log-walks and their extra work, and by providing breakdowns of fast and slow commits and aborts.

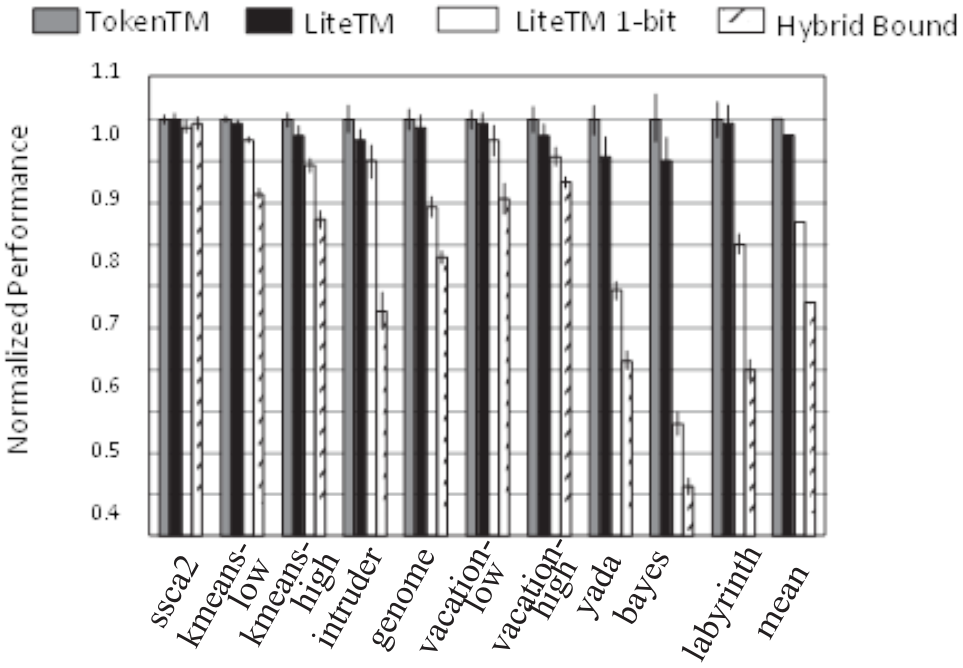

Fig 2.2 LiteTM performance

Table 2.4 Benchmarks

Benchmark	%xact time	#instr/xact	xact length	#aborts / # commits	contention	TokenTM vs. LogTM-SE	TokenTM vs. Single
Ssca2	13	13	short	0.01	low	1.01	5.6
k-means-low	7	106	short	0.03	low	0.99	3.5
k-means-high	11	106	short	0.07	low	1.02	3.1
Intruder	56	187	short	1.25	med.	1.11	2.6
Genome	61	1209	med.	0.11	low	1.05	4.0
Vacation-lo	92	1640	med.	2.78	high	1.38	4.4
Vacation-high	92	2218	med.	2.56	high	1.46	4.1
Yada	97	5715	long	1.17	med.	2.36	2.5
Bayes	91	39213	long	1.82	high	2.19	2.7
Labyrinth	99	147515	long	3.86	high	2.72	2.0

2.6.1. LiteTM Performance

In Figure 2.2, I compare LiteTM's performance to that of TokenTM. Both the LiteTM 1-bit variant (Section 4.8) and HTM-STM hybrids have only one bit per L1 block in L2 and memory which is half the state overhead of LiteTM. (Hybrids need one bit to ensure isolation among hardware and software transactions, and among transactions and non-transactions (i.e., strong atomicity) [5].) Therefore, I include these two TMs in this comparison with one twist: Because I do not have access to a fully-optimized hybrid,

I use a hybrid variant which provides an upper bound on hybrid's performance. In hybrids, transactions switch from HTM to STM when a transactional block is evicted. In my variant, hardware transactions use TokenTM, whereas software transactions perform a single extra memory write to a globally-shared hash table whenever a new token is obtained (i.e., the first transactional access to a block). I do not impose any other overheads on software transactions upon commit or abort (e.g., to clear the hash table). Because software transactions perform at least one write to a globally-shared hash table to update read sets and write sets for conflict detection (in reality, STMs incur more memory accesses to update other structures such as per-transaction private read/write set and undo log [1]), my variant provides an upper bound on hybrids' performance.

Figure 2.2 shows the performance of LiteTM, LiteTM 1-bit variant, and hybrid upper-bound variant normalized to that of TokenTM. The X axis shows the benchmarks in the order of primarily increasing transaction length and secondarily increasing contention (Table 2.4). This ordering clearly shows trends across benchmarks. LiteTM incurs 4% average degradation over TokenTM with the worst degradation of about 10% for bayes. LiteTM-1-bit incurs significantly higher average (24%) and worst case (72%) degradation than LiteTM. LiteTM-1-bit requires numerous self and all log-walks to distinguish between transactional reads and writes, and single and multiple readers, whereas LiteTM uses its two state bits to make the distinction (Section 2.4.4). Hybrid-bound incurs 44% average degradation over LiteTM. This degradation comes from hybrids' software conflict detection on every access of software transactions whereas LiteTM incurs log-walks only for L1-evicted blocks, as mentioned in Section 2.1. Given that both LiteTM-1-bit and hybrids need one state bit per block, these results justify the use of two state bits in LiteTM. Combining TokenTM's speedups over LogTM-SE from Table 2.4 and LiteTM's slowdowns over TokenTM, LiteTM performs 36%, on average, better than LogTM-SE, maintaining TokenTM's performance advantage over LogTM-SE.

Finally, LiteTM's degradation mostly increases as transaction length and contention increase across benchmarks (from left to right). Longer transactions and higher contention result in more evictions and conflicts which require the use of thread identifier

and read-sharer count triggering more log walks and hence incurring higher performance degradation in LiteTM. The main outlier is labyrinth which degrades less than yada and bayes despite having higher contention and longer transactions. Because of labyrinth's extremely long transactions and high contention (Table 2.4), TM's optimistic execution incurs significant overhead — aborts and back-off time for relaunch after aborts (Section 2.5). I tried turning off back-off which led to starvation due to excessive aborts. Compared to this high overhead, LiteTM's additional overhead is relatively small, resulting in less degradation for labyrinth as compared to those for yada and bayes. The trend of increasing degradation from left to right also holds, albeit with more outliers, for LiteTM-1-bit and hybrid-upper-bound.

I ran LiteTM on 16 cores with a slight implementation variant of the W-approximation. The degradations over TokenTM were statistically similar to those of the 8-core runs (not shown). I also experimented with adding R, W, R', and W' bits to LiteTM and saw less than 5% improvement in performance.

Table 2.5 Log-walks

Benchmark	%miss to own block	%out-of-cache abort	%conflict on L2 rd-shared	%false abort	Self + all log-walk/#commit	Self log length	All log length
Ssca2	0.04	0	0	0	~0	2	0
k-means-low	0.16	0	0	0	~0	3.3	0
k-means-high	0.24	0	0	0	~0	4	0
Intruder	0.4	0.04	0.04	0	~0	22	34
Genome	0.39	0	0.65	2.5	0.02 + ~0	17	27
Vacation-lo	0.01	0	0.36	0	~0	28	36
Vacation-high	0.05	0.03	0.4	0	0.02 + 0.01	38	54
Yada	2.3	0.8	0.5	0.9	0.3 + ~0	97	102
Bayes	6	1.1	1.9	0.3	3.9 + 0.08	162	327
Labyrinth	15	5.6	2.5	0.1	58 + 0.94	272	1408

2.6.2. LiteTM Performance Analysis

The key explanation for LiteTM performing close to TokenTM is as follows. LiteTM's self and all log-walks with lazy clearing compensate for the loss of information in terms of separation of reads and writes, read-sharer count, and thread identifier (Table 2.2). LiteTM's log-walks not only increase the amount of compute work, but also (1) replace transactional data with log data causing more slow-commits (commit after eviction) and slow-aborts (conflict on block evicted by multiple read sharers); and (2) stretch transactions' life times and induce more conflicts (i.e., log-walks delay token release keeping blocks in transactional state longer). However, as stated before, the cases requiring the information and, consequently, the log-walks are uncommon. Hence, the

log-walks' performance overhead is low. Accordingly, I quantify how often these pieces of information are needed and the number of log-walks in Section 2.6.2.1, and the number of extra slow commits and aborts in Section 2.6.2.2.

Table 2.6Commits and aborts

Benchmark	TokenTM			LiteTM		
	#aborts/ #commits	%slow commits	%slow aborts	#aborts/ #commits	%slow-commits	%slow-aborts
Ssca2	0.01	0.4	0	0.01	0.5	0
k-means-low	0.03	2.1	0	0.04	2.7	0
k-means-high	0.07	2.87	0	0.08	2.9	0
Intruder	1.3	31	0	1.4	33	0.04
Genome	0.1	3.1	0	0.1	9.2	0
Vacation-lo	2.8	38	0	3.1	54	0
Vacation-high	2.6	33	0	2.9	56	0.03
Yada	1.2	15	~0	1.1	30	0.8
Bayes	1.8	17	~0	2.7	38	1.1
Labyrinth	3.9	26	0	5.3	95	5.6

2.6.2.1. Log Walks

Recall that (1) TokenTM uses the thread identifier to recognize a transaction's own blocks that either are fused in another cache (Section 2.3.1) or are evicted from L1 (Section 2.3.3), and to identify the transactions involved in conflicts on evicted blocks (Section 2.3.3); (2) TokenTM uses the read-sharer count to determine conflicts on read-shared blocks (Section 2.3.3); and (3) LiteTM approximates W as a combination of modified state and T (Section 2.4.1). Table 2.5 quantifies how often (1) these uses occur which require self and all log-walks with lazy clearing, and (2) W approximation is

inaccurate and induces false aborts. The table shows the misses to a transaction's own blocks as a percent of all transactional misses (%miss to own block), the out-of-cache aborts due to conflicts on evicted blocks as a percent of all aborts (%outof-cache abort), the conflicts on read-shared L2 blocks which have been evicted from more than one sharer's L1 — true conflicts and false conflicts requiring lazy clearing — as percent of all conflicts (%conflict on L2 rd-shared), and the false aborts due to W-approximation as percent of true aborts (%false abort).

I see that % miss to own block and % out-of-cache aborts, which together correspond to use of thread identifier in TokenTM, are low, confirming that repeated misses to the same blocks and conflicts on L1-evicted blocks are rare. % conflicts on L2 rd-shared block, which corresponds to the use of read-sharer count in TokenTM, is low, confirming that conflicts involving multiple L1- evicted read-shared blocks are rare. For the most part, only the benchmarks with higher contention and longer transactions — yada, bayes, and labyrinth (Table 2.4) — have significant quantities, as expected. LiteTM replaces the identifier and count, respectively, with self log-walks and all log-walks with lazy clearing, which, consequently, are infrequent. I also see that % false aborts is low implying that the W-approximation is rarely inaccurate. The number of self and all log-walks per committed transaction (#self + all log-walk / #commit) reconfirm that the self and all log-walks are mostly infrequent with the self log-walks occurring more often than the all log-walks. labyrinth with its numerous self log-walks is an exception. However, labyrinth's log-walks do not degrade performance as they are amortized over the extremely long transactions (Table 2.4). Across the benchmarks, however, each instance of the log-walks scan many addresses as shown by the number of addresses per self log-walk and all log-walk, respectively, in the last two columns in Table 2.5 (self log length and all log length). The high log-walk volume, especially of the more-often-occurring self log-walks, increases the compute work, degrading LiteTM's performance. The volume also results in evictions inducing extra slow-commits and slow-aborts, and stretches transactions' life times inducing more aborts. I analyze these effects next.

2.6.2.2. Commits and aborts

Table 2.6 compares TokenTM and LiteTM in terms of the ratio of number of all aborts to number of all commits (#aborts / #commits), the percent of slow commits over all commits (%slow-commits), and the percent of slow aborts over all aborts (%slow aborts). Comparing the abort-to-commit ratios in TokenTM and LiteTM, the benchmarks with relatively more log-walks (#self + all log-walk / #commit in Table 2.5) — bayes and labyrinth — have more aborts (the two TMs have the same number of commits). The increase in aborts is due to the stretching of the transactions' lifetimes. yada is an exception with many log-walks but fewer aborts. Instead of more aborts, yada incurs more back-off delay for relaunch after abort (Section 2.5). k-means low, vacation low, and vacation high are exceptions in the opposite direction: relatively few log-walks but many more aborts. The increase in the percent of aborts appears to be large for k-means low because the absolute number of aborts is small, as confirmed by the small performance degradation (Figure 2.2). In vacation low and vacation high, the extra aborts are due not to the log-walks but include the aborts of all-but-one L1-resident read-sharer involved in a conflict (Section 2.4.2). These all-but-one aborts serialize the read-sharer inducing even more conflicts in these high-contention benchmarks (#aborts/#commits in Table 2.4).

Comparing the percent of slow-commits in TokenTM and LiteTM, the benchmarks with relatively more log-walks — yada, bayes, and labyrinth (Table 2.5) — evict many transactional blocks leading to slow commits which contribute to their performance degradation (Figure 2.2). Though labyrinth has the largest increase in slow commits, its basic optimistic execution overhead, as discussed in Section 2.6.1, dwarfs its slow-commit overhead. Vacation-low and vacation-high are exceptions which have more slow commits but relatively few log-walks (#self+all log-walk / #commit in Table 2.5). These two benchmarks' extra aborts (discussed above) imply more conflicts which lead to more slow-commits because transactions that survive conflicts undergo slow-commits (Section 2.3.2). However, because the aborts far outnumber the commits in these benchmarks (Table 2.6), the abort overhead dwarfs the extra slow-commit overhead so that overall

performance degradation is not much (Figure 2.2). Finally, I see that slow-aborts are infrequent in both TokenTM and LiteTM.

2.6.2.3. Sensitivity to number of busy buffers

Recall from Section 2.4.4 that LiteTM uses busy buffers and a busy counter to handle races between log-walks for lazy clearing and transactional accesses. In Figure 2.3, I show LiteTM's performance varying the number of buffers as four (same as Figure 2.2, see Table 2.3) and zero normalized to that of TokenTM. With no buffers, a non-zero busy counter implies that a lazy clearing is underway and flags all misses during a lazy clearing to be serialized after the lazy clearing, under the conservative assumption that the misses are to the block being lazy-cleared.

2.7. Conclusions

To allow transactional state to exceed cache capacity, recent HTMs (e.g., TokenTM, VTM, OneTM-concurrent) employ per block thread identifiers and sharer counts. The resulting overhead can be high, especially if the state is held in stolen ECC bits. LiteTM cuts the overhead by decoupling the detection of conflicts (done in hardware) from the identification of conflicting transactions (done in software using transactional logs, in the uncommon case). LiteTM compensates for the lost information via the following novel ideas: (1) approximating W for L1-resident blocks, (2) lazy clearing of transactional state in L2 and memory, (3) self log-walk to identify a transaction's own blocks. LiteTM requires just 2 bits per block in L1, L2, and memory. Experiments show that LiteTM reduces TokenTM's state overhead by about 87% while performing within 4%, on average, and 10%, in the worst case, of TokenTM. By reducing transactional state overhead while maintaining performance, LiteTM lowers the barrier for adoption of HTMs in real products

3. WAIT-N-GOTM

3.1. Introduction

Most TMs optimistically allow concurrent transactions, detecting conflicts when a transaction attempts to read or write a data block written by another concurrent transaction. While STMs use software to detect conflicts, HTMs piggyback conflict detection on cache coherence and hence are faster. Therefore, I focus on HTMs though my ideas are applicable to STMs.

Upon detecting a conflict, current HTM proposals employ one of three conflict-resolution policies: (1) Always-abort: abort one or more of the conflicting transactions (e.g., [3,7,12,27,36]). (2) Always-wait: force one or more of the conflicting transactions to wait for the other(s) to commit or abort, without (e.g., [22,28]) or with (e.g., [29]) thread pre-emption. (3) Always-go: allow the conflicting transactions to proceed by forcing commits in the conflict induced dependence order (e.g., DATM [37]) or by re-executing the backward slice of the conflicting accesses to repair corrupted memory state (e.g., RETCON [30]). Because such repair is hard in the general case, RETCON is limited to simple cases such as counter increments. Another always-go variation, lazy conflict resolution, delays conflict detection until commit, and hence allows transactions to proceed past as-yet-unknown conflicts (e.g., [16]). In general, both always-wait and always-go may incur cyclic waiting and dependencies, respectively, which are resolved via aborts.

While each conflict resolution policy has advantages, they all degrade performance in the presence of contention. Always-abort incurs many aborts even for acyclic dependencies which can be resolved by committing in the dependence order (i.e., producer before consumer). Always-wait aborts less often but incurs long waiting for conflicts to disappear via commits or aborts, disallowing producer-consumer concurrency

for acyclic dependencies. Always-go suffers fewer aborts and less waiting for acyclic dependencies. However, the policy incurs late aborts due to cyclic dependencies formed after proceeding past many conflicts, resulting in much lost work. Further, frequent aborts are expensive as TMs typically use randomized back-off to reduce the chances of future conflicts. Improving TM performance is important given that many companies have built or considered building HTM support into their products, including IBM [41], Intel [33], AMD [31], Oracle [34], and Azul [32].

I along with my colleague Gwendolyn Voskuilen and adviser T. N. Vijaykumar propose Wait-n-GoTM (WnGTM) in [WNG] to address the above performance problems: (1) always-abort and always-wait limit concurrency while always-go incurs late aborts; and (2) none of the existing policies avoid aborts upon cyclic dependencies. WnGTM is based on the guiding principle that to achieve high performance, conflict resolution policies should increase concurrency while avoiding cyclic aborts. My primary contribution is a new conflict resolution policy, called Wait-n-Go (WnG), which achieves high performance by serializing in-flight, cyclically-conflicting transactions for the right amount of time to avoid many cyclic aborts and then allowing the transactions to proceed past conflicts to increase concurrency. For acyclic dependencies, WnG defaults to always-go to avoid aborts and waiting.

I make the key observation that most cyclic dependencies, and hence aborts, are caused by a few, heavily-read-write-shared data cache blocks. Multiple accesses to one or more such delinquent blocks in one transaction are cyclically interleaved with accesses from another transaction, resulting in dependence cycles. I call the section of transactional code comprising all the accesses to a particular delinquent block a cycle inducer section (CIST). (Transactions accessing multiple delinquent blocks have multiple CISTs.) I also observe that CISTs predictably repeat due to systematic program patterns, such as simple operations like enqueue and dequeue of a work queue and update of auxiliary, book-keeping counters, as well as complex computations like general graph insertion and deletion. Such systematic patterns expose the lack of robustness of previous approaches, and result in performance degradation. I specifically note that while lazy conflict resolution implicitly employs always-go and naturally avoids some aborts upon

acyclic dependencies, lazy resolution cannot avoid late aborts due to CISTs, just as always-go with eager resolution (e.g., DATM) cannot.

WnG avoids cycles by serializing the CISTs of conflicting transactions. To this end, I propose a hardware CIST predictor that predicts when a transaction needs to be serialized and for how long to avoid aborts. The predictor flags the beginning of a CIST so that the (successor) transaction waits for predecessor transactions to exit the CIST (the "wait" in WnG). A successor waiting until the predecessors commit would result in extra waiting (similar to always-wait). Instead, the predictor flags the end of the CIST so that a predecessor can signal the waiting successors to proceed even if the predecessor has not yet committed (the "go" in WnG). The predictor can handle CISTs with arbitrary control and data flow. Because only a few blocks are delinquent, small prediction tables suffice (e.g., 32 entries per core); and because the delinquent blocks are highly predictable, WnG's waiting is accurate. Conflicting transactions are serialized only through the CISTs, and execute concurrently both before and after the CISTs. Consequently, WnGTM orders commits in the predecessor-successor order to maintain serializability, similar to DATM. This execution schedule satisfies my goal of increasing concurrency while avoiding cyclic aborts. Unlike RETCON [7], WnG can handle CISTs comprising arbitrary computation. Thus, WnG achieves good performance despite systematic cyclic dependencies, and exhibits robustness. Serializing in-flight, conflicting transactions, as WnG does, is inevitably involved. I manage this complexity by dividing the labor between hardware and software, both for the WnG policy and the mechanisms for waiting at and proceeding past conflicts. On the policy side, I use the hardware solely for the simpler task of CIST prediction and push to software exception handlers the burdensome parts of serialization. On the mechanism side, the hardware handles the simple cases of events that are captured by coherence while the software handles the more difficult remaining cases. Specifically, I use three mechanisms proposed by Voskuilen et al. in [42] to extend previous HTMs. The first mechanism generalizes the multi-reader, single-writer abstraction (i.e., multiple reader transactions or a single writer transaction per cache block) which does not order transactions, as used by many previous HTMs, to a multi-reader multi-writer paradigm (i.e., multiple transactions reading and

writing the same block concurrently) like DATM which orders transactions. To achieve this ordering, Voskuilen et al. use a transactional (not coherence) state, conflict state, to identify blocks involved in one or more conflicts. The second mechanism proposes order-capture which captures the predecessor-successor ordering in hardware in the simple, common case of conflicts through the L1 cache and in software in the more difficult, uncommon case of conflicts through the L2. Finally, Voskuilen et al. adapt a hardware timestamp scheme from [39] to detect cyclic dependencies and waiting, without relying on broadcasts and degrading scalability like DATM. Voskuilen et al's timestamp scheme is independent of the WnG policy ensuring policy-mechanism separation. A key aspect of these mechanisms is that they avoid changing coherence by leveraging TokenTM [7].
In summary, the contributions of this chapter are:

• Wait-n-Go conflict resolution policy which increases concurrency while avoiding many cyclic aborts by making conflicting transactions wait for the right amount of time before proceeding even if predecessor transactions have not yet committed;

• Hardware CIST predictor to serialize only the CISTs while allowing concurrent execution before and after the CISTs;

• In 16-core simulations of STAMP, WnGTM achieves, on average, 46% and 28% speedups over always-abort (TokenTM) for the higher-contention benchmarks and all the benchmarks, respectively, with low-contention benchmarks remaining unchanged, compared to always-go (DATM) and always-wait (LogTM-SE [28]), which perform worse than and 6% better than TokenTM, respectively. The rest of the chapter is organized as follows. Section 3.2 describes my Wait-n-Go policy and Section 3.3 describes Wait-n-GoTM's mechanisms. I discuss related work in Section 3.4. I describe my experimental methodology in Section 3.5 and present my results in Section 3.6. I conclude the chapter in Section 3.7.

3.2. Wait-n-Go Conflict Resolution

Because the WnG policy is applicable to any TM, I present the policy in this section without referring to any specific TM. Fig 3.1 illustrates eager always-go (a) and WnG (b) for both acyclic and cyclic conflicts. In the acyclic conflict via non-delinquent data cache block A, transaction T2 writes A after T1 has read A (time grows downwards). This

conflict causes always-abort to abort and always-wait to wait, while always-go (shown on the left) proceeds past the conflict. WnG defaults to always-go for acyclic conflicts and also proceeds past the conflict (on the right). In the cyclic conflict via delinquent cache block B, T1 reads B, T2 reads and writes to B, and then T1 writes to B.

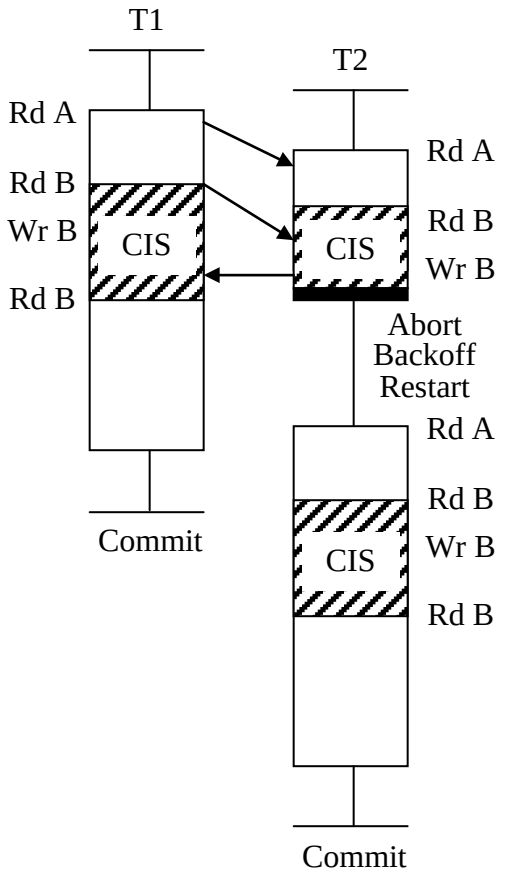

Fig 3.1Execution of a CIST for delinquent data cache block B
(a) with always-go and (b) with WnG

This cyclic access pattern causes an abort under both always-abort and always-go (on the left). If not for the initial acyclic conflict, which causes T2 under always-wait to wait (excessively) for T1 to commit, the cyclic accesses would also cause always-wait to abort due to cyclic waiting. However, with WnG, on the right, T2 waits until T1 completes B's CIST and then T2 enters the CIST and executes in parallel with the remainder of T1. Thus, T2 avoids aborts, lost work, and time spent in back-off, while achieving concurrency with T1. Figure 3.1 depicts always-go for eager systems. I clarify that lazy conflict resolution, which implicitly uses always-go, avoids some but not all aborts due to acyclic dependencies. Because lazy resolution implies lazy version management (eager versioning is not possible), acyclic dependencies do not restrict commit order and may cause aborts. For example, in a simple, acyclic write-to-read conflict between two transactions, the reader obtains the old value before the write due to lazy versioning. If the reader reaches commit before the writer (by chance), there is no abort; but if the writer reaches first then the reader has read a stale value and must abort. In contrast, always-go with eager conflict resolution (e.g., DATM) forces the commit order to be in dependence order (either reader first or writer first) and no abort occurs. In addition to such acyclic aborts, lazy resolution cannot avoid cyclic aborts due to CISTs such as the one in Figure 3.1. The WnG policy consists of two components: (1) a CIST predictor to learn delinquent blocks and CIST boundaries, and (2) using the predicted CIST information to serialize the CISTs.

3.2.1. Learning the CISTs

To serialize CISTs, the CIST predictor must learn (1) the identities of delinquent blocks, and (2) the boundaries (i.e., start and end) of the corresponding CISTs. For identities, I leverage the fact that only a few delinquent blocks cause the majority of cyclic aborts (e.g., 3 blocks cause 95% of aborts in intruder). Thus, block addresses causing aborts are recorded in a per-core hardware CIST prediction table (CPT). I identify repeat offenders as delinquent by incrementing a per table-entry saturating counter whenever the corresponding address triggers an abort. Figure 3.2 illustrates the CPT. Incrementing the counters upon conflicts rather than aborts would lead to excessive waiting due to many blocks being labeled as delinquent. Because a few blocks cause

most aborts, the CPT is small and can be searched in parallel with the L1 cache (e.g., 32 entries per core). As in other hardware prediction schemes, CIST learning occurs throughout execution. To avoid cycles, a CIST must include all accesses to a delinquent block within a transaction (e.g., the CISTS in T1 and T2 in Figure 3.1). Otherwise, a delinquent block might correspond to multiple CISTs, and those CISTs could be arbitrarily interleaved across threads. Thus, by construction (i.e., not a program property), there is a one-to-one correspondence between delinquent blocks and CISTs. To determine CIST boundaries, I assume that a CIST starts at a transaction's first dynamic access to a delinquent block and ends at the last access to the block. While the first access can be determined with 100% accuracy, the last access requires determining when a transaction will no longer access a particular block. I observe that last accesses in previous dynamic instances of a transaction predict the last access in subsequent instances and record the instruction count (from transaction begin) at which the last access occurred, called CISTend, in the CPT (see Figure 3.2). To account for variations across instances, I conservatively use the maximum of the instruction counts among instances, called max-count. Thus, hardware updates the CISTend whenever a delinquent block is accessed beyond its current CISTend. Alternative CISTend metrics, such as PC and the difference between successive delinquent block accesses, tend to be inaccurate due to varied control flow. While max-count could cause excessive waiting due to a large count in an infrequent control-flow path, my results show that max-count is accurate (Section CIST statistics3.6.2). Still, incorrect CISTends and slow learning of delinquent blocks may occur and lead to cyclic dependencies which are detected by my timestamps (TS in Figure 3.2) and result in aborts (Section 3.3). Because blocks may cease to be delinquent over time due to lack of accesses, WnG unlearns by decrementing a block's counter at commit when the block associated with a transaction is not accessed in the transaction (aborts do not decrement as an abort may occur before the block is accessed). To capture this association, each CPT entry also holds a unique static transaction identifier assigned by the programmer or compiler (TxID in Figure 3.2). Further, although a CIST starts at a transaction's first access matching an address in the CPT, WnG records the predicted CIST start instruction, CISTbegin, for the separate purpose of handling overlapping

CISTs (Section 3.2.2.2). Analogous to CISTend, CISTbegin uses the minimum instruction count. I will discuss the remaining CPT field (read-only counter) and other state in Figure 3.2 — L1/L2 transactional state, timestamps (TS), and dynamic transaction (Tx) count — later in Section 3.2.2.5 and Section 3.3.

Finally, I accelerate delinquent block and CISTend learning via an aggressive gossip mechanism. Conflicting transactions use gossip to, whenever possible, force immediate saturation of the conflicted block's CPT counter if the block is delinquent for even one of the transactions. Additionally, transactions with the same static identifier exchange CISTends and set the matching CPT entry's CISTend to the largest value. A block's delinquency affects all accessing transactions, irrespective of static identifier, so saturation is gossiped among all conflicting transactions. However, CISTend is static transaction specific and is therefore gossiped only to same-static-identifier transactions. Because conflicting accesses usually generate coherence traffic among the conflicting transactions (conflicts involve writes), gossip information can be piggybacked on coherence. Conflicts via evicted blocks do not generate coherence messages among conflicting transactions so gossip does not occur. Finally, while normal delinquent block learning occurs only upon aborts (for precise learning), gossip occurs upon conflicts as well (to rapidly disseminate learned information).

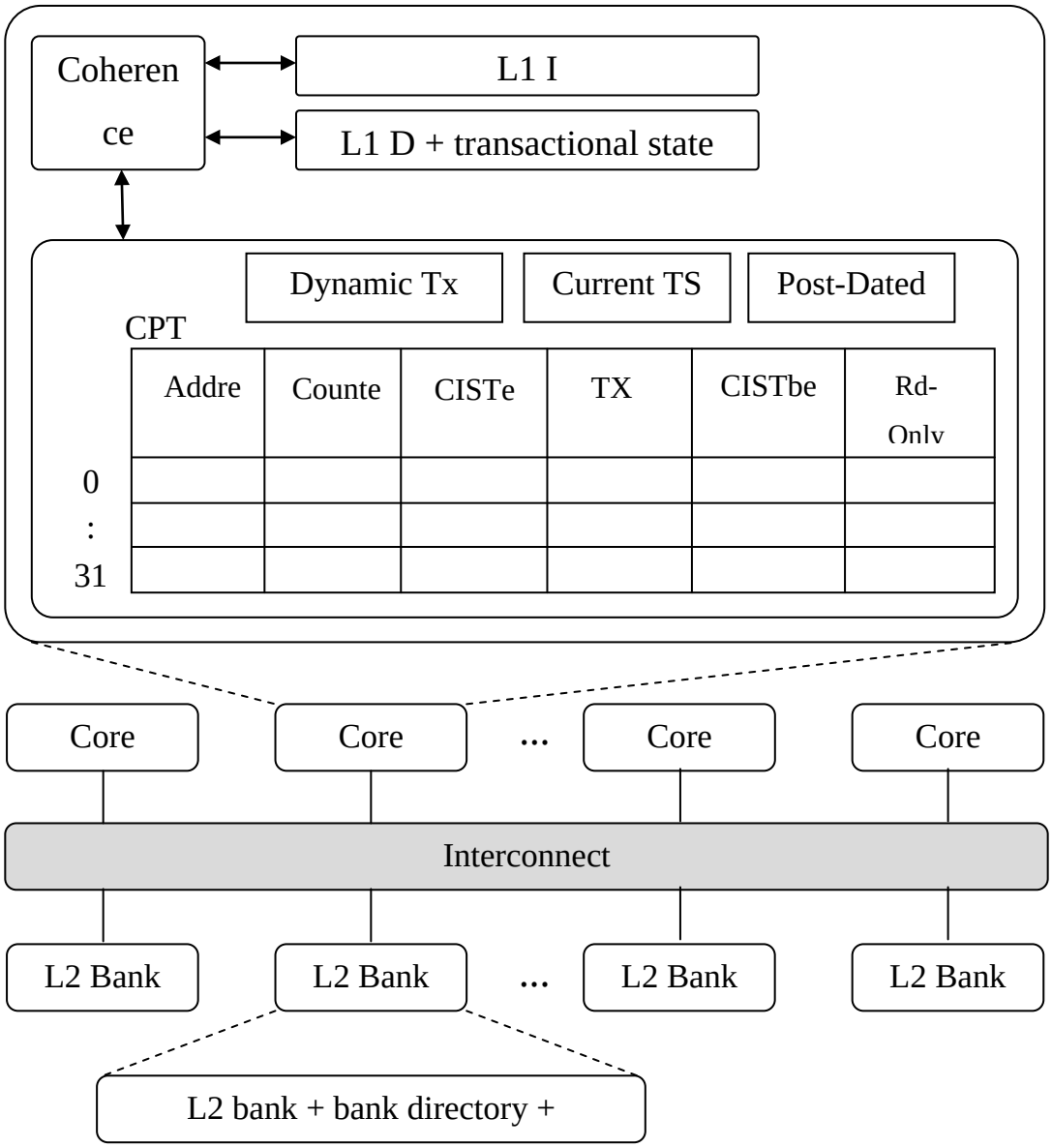

Fig 3.2 WnGTM hardware

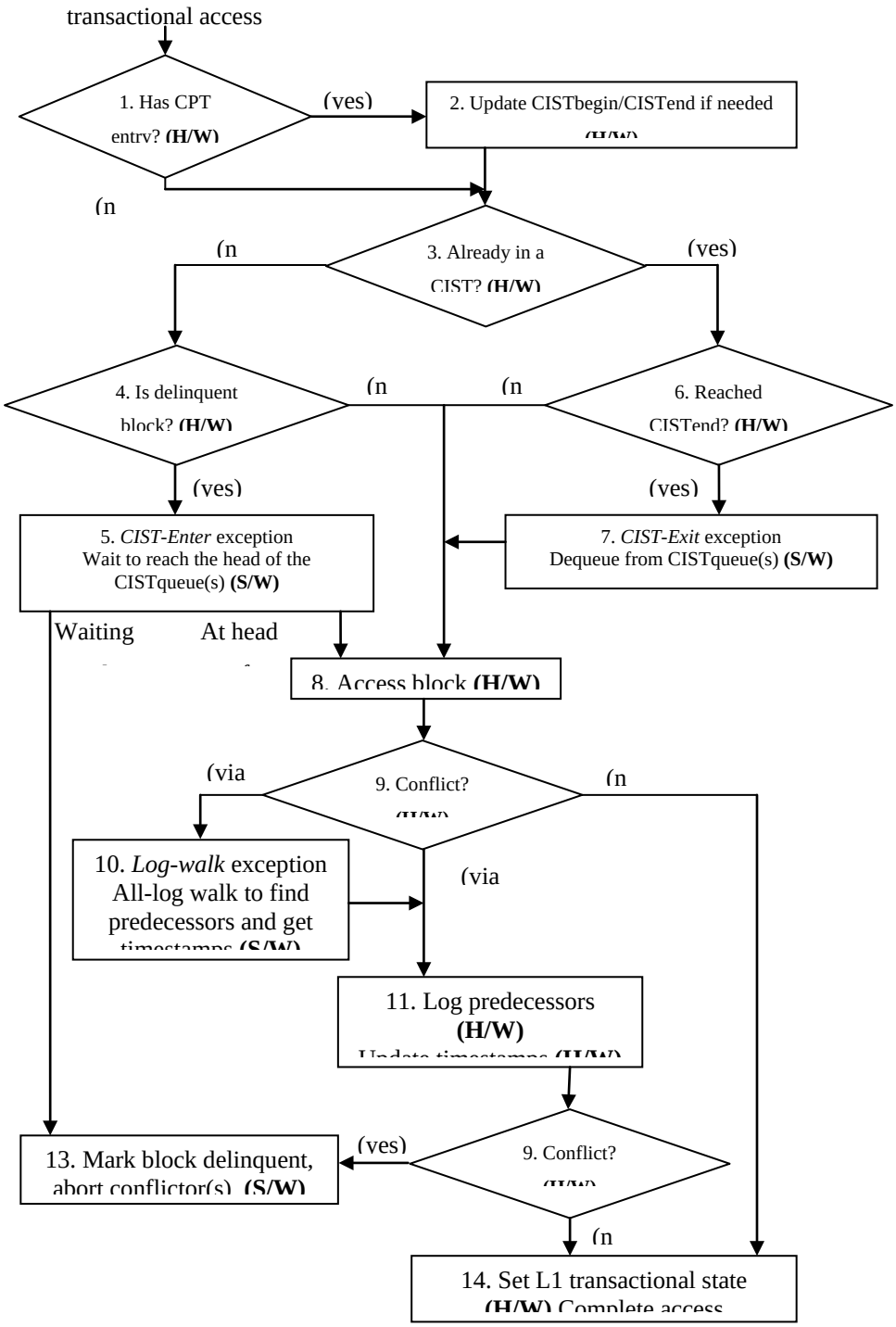

Fig 3.3WnGTM access flowchart

3.2.2. Serializing the CISTs

The WnG conflict resolution policy serializes CISTs. I first cover the simple case of serializing non-overlapping CISTs and then cover fully- or partially-overlapping CISTs. Finally, I describe a few optimizations for my serialization policy.

3.2.2.1. Non-overlapping CISTs

WnG defaults to an always-go policy (similar to DATM). Accesses to non-delinquent blocks (i.e., the block's address is not in the CPT or the block's counter is not saturated) proceed even if they cause conflicts. This choice causes a few delinquent block aborts which help to bootstrap learning, and allows non-delinquent blocks to form acyclic dependencies which do not cause aborts. Timestamps detect any cycles via non-delinquent blocks, similar to those caused by CISTend mispredictions (Section 3.2.1). Fig 3.3 gives the flowchart of an access in WnGTM. Every transactional access searches the CPT (Figure 3.3, step 1) and updates the table as necessary (Figure 3.3, step 2). Recall that accessing a delinquent block (Figure 3.3, step 4) is synonymous with entering a CIST. (Step 3 pertains to overlapping CISTs and is discussed in the following section.) If the delinquent block access causes a cycle (typically when CIST learning is incomplete), the transaction aborts and increments the block's CPT counter as described in Section 3.2.1 (Figure 3.3, step 13); otherwise the access triggers a CIST-Enter exception (Figure 3.3, step 5). I show the precise steps taken during a CIST-Enter exception in Figure 3.4. The exception serializes threads attempting to enter the CIST by enqueuing the threads in their timestamp order in a per-block CISTqueue (Figure 3.4, steps 1, 2). Though timestamp-ordered enqueuing increases the computation overhead, the ordering avoids cyclic dependencies and, hence, expensive aborts. The thread at the head of the CISTqueue executes the CIST (Figure 3.4, step 4) and threads are removed as they exit the CIST. Each thread's exception handler simply waits to reach the head of the queue (i.e., the waiting is in software and does not require coherence changes). Figure 3.5 shows the CIST serialization between two transactions T1 and T2. Upon accessing delinquent block A, T1 raises a CIST-Enter exception and is placed in block A's CISTqueue (back to Figure 3.3, step 5). When T2 accesses A, T2 raises a CIST-Enter

exception and waits in A's CISTqueue until T1 exits. To flag CIST exit, upon reaching the CISTend instruction count for the CIST, the thread executing the CIST (T1) raises a CIST-Exit exception (Section 2.1) and is dequeued from A's CISTqueue (Figure 3.3, steps 6, 7). Because transactions raise exceptions at both CIST entry and exit, the exception overhead may be high for short transactions. An option would be to serialize the entire transaction instead of only the CIST. Then, serialization would occur as part of transaction begin, which can be a function call instead of an exception. However, I tried this optimization and did not see any benefits. Finally, the remaining steps, 8 - 12 and 14, refer to WnGTM's mechanisms and are explained in Section 3.3. Because CISTqueues are per-block, CISTs corresponding to different delinquent blocks may be executed concurrently, improving performance. While dependence cycles may occur if threads' transactions execute multiple CISTs in different orders, most transactions have a single CIST or a single set of overlapping CISTs (covered in the next section) and do not face this issue. For the remaining cases, my timestamp-ordered enqueuing increases the likelihood of (but does not guarantee) the same order for all the CISTs. Finally, because waiting induces dependence between the waiting thread and the thread being waited for, cyclic waiting is possible. My timestamp scheme detects these waiting cycles, as shown in Figure 3.4, steps 2-3, and aborts the transaction (Figure 3.3, step 13), just as for data dependence cycles.

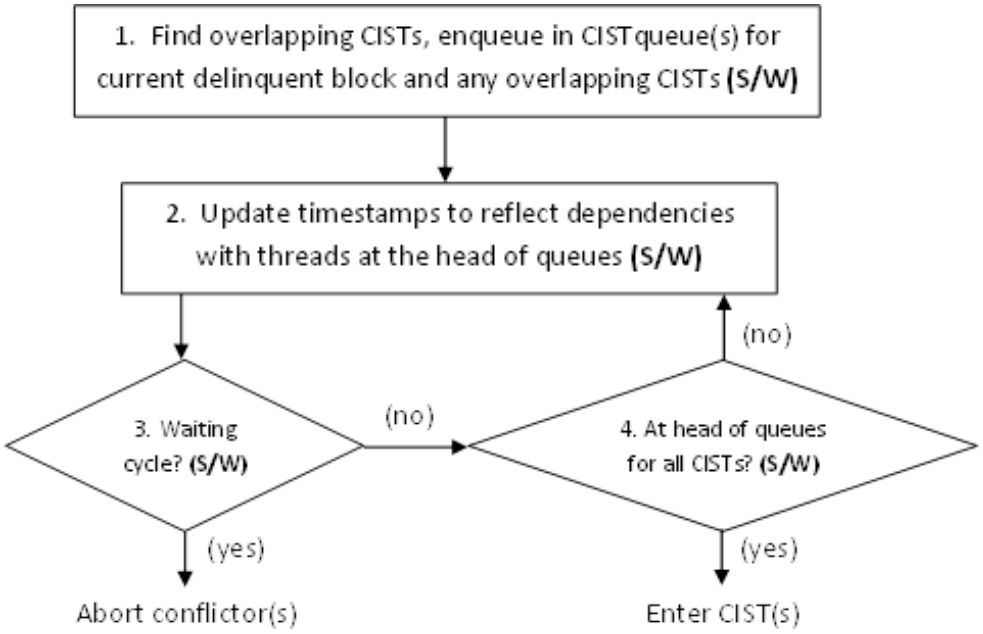

Fig 3.5 CIST enter exception

3.2.2.2. Overlapping CISTs

Overlapping CISTs occur when a transaction executing a CIST for a delinquent block accesses a different delinquent block before exiting the first CIST. If these overlapping CISTs were treated like non-overlapping CISTs, with each causing a CIST-Enter and CIST-Exit exception, the first CIST would incur a long delay due to the exceptions; I observe overlapping CISTs are not infrequent (e.g., in intruder, 27% of transactions have overlapping CISTs). Collapsing overlapping CISTs into a single CIST does not solve the problem because the collapsed CIST would be associated with the earliest accessed delinquent block. This block may differ among threads, causing threads to wait at different CISTqueues for the overlapping CISTs (analogous to acquiring different locks in different threads). Consequently, threads would not wait for each other and would incur cyclic dependencies. For example, in Figure 6, transaction T1 would

wait in block B's CISTqueue while T2 would wait in block A's CISTqueue. To avoid the above issues, I force a thread to wait simultaneously for all overlapping CISTs and allow a thread to enter the CISTs only when the thread is at the head of the queue for all such CISTs (analogous to acquiring all locks together). To identify overlapping CISTs, when a thread raises a CIST-Enter exception at the first delinquent block access, the exception handler walks the CPT and uses the CISTbegin and CISTend fields to determine which other CISTs may overlap with the accessed block's CIST. (I track CISTbegin for this purpose as stated in Section 3.2.1.) The handler then enqueues the thread in timestamp order in the CISTqueues for each of the overlapping CISTs (Figure 3.4, steps 1, 2). Note that CIST-Enter exception (Figure 3.4) is identical for both the non-overlapping and overlapping case, except that in the overlapping case, a thread waits in multiple CISTqueues. As with non-overlapping CISTs, the handler uses timestamps to detect any cycles in the CISTqueues and aborts. Because it is serialized up front, the transaction does not raise further CIST-Enter exceptions upon encountering overlapping CISTs (Figure 3.3, step 3). Thus, in Figure 3.6, T1 raises a single CIST-Enter exception to access the overlapping CISTs for blocks A and B. To exit overlapping CISTs, a transaction raises a CIST-Exit exception upon reaching the latest of the overlapping CISTs' CISTends, hence, in Figure 3.6, T2 must wait for T1 to finish executing both A's and B's CIST before T2 may enter A's CIST.

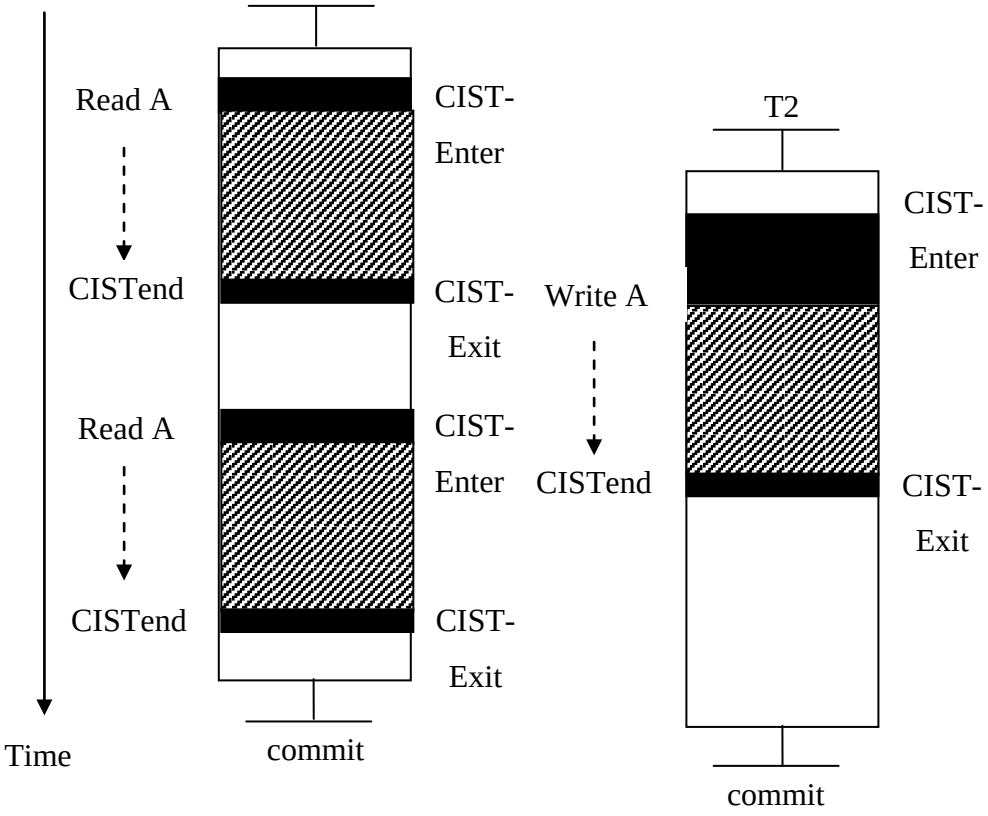

Fig 3.6 Non-overlapping CISTs

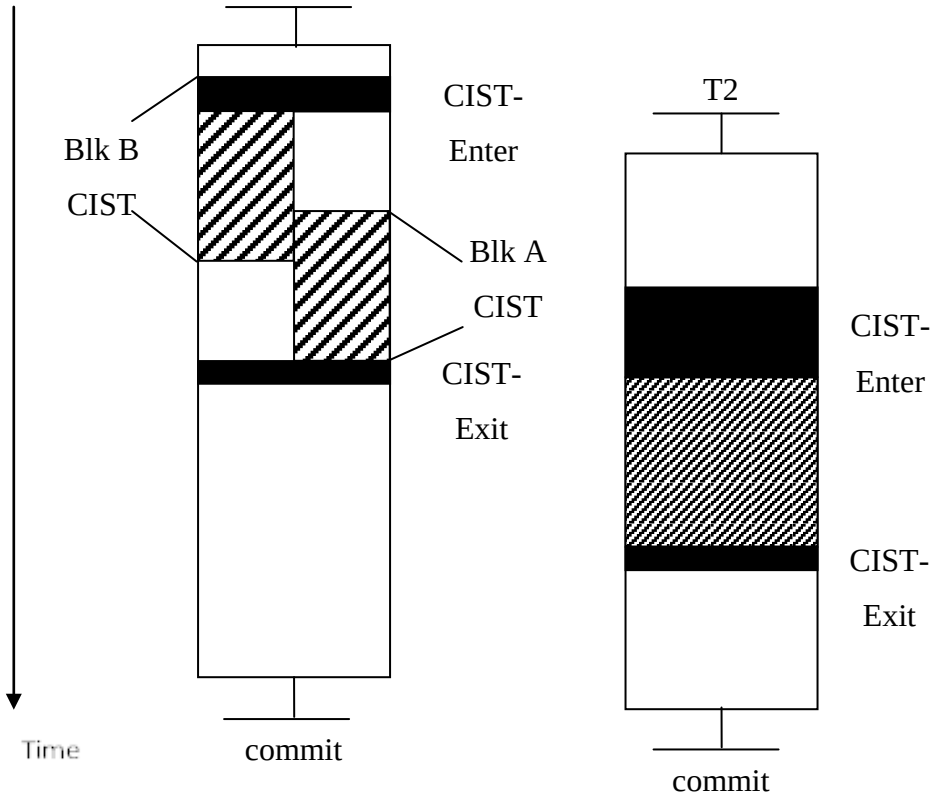

Fig 3.7 Overlapping CISTs

3.2.2.3. Cycles through multiple acyclic conflicts

Thus far I have discussed cycles formed by multiple accesses to one block (i.e., a CIST) and my CIST prediction targets this case. However, I observe that it is also possible for cycles to form due to acyclic conflicts on multiple blocks, each accessed in different orders in different transactions. If such cycles were systematic, the acyclically conflicting blocks would be learned as delinquent using WnG's learning but serializing the acyclically conflicting blocks' non-overlapping CISTs would not avoid the cycle. To handle this case, CIST prediction could be extended to merge accesses to multiple delinquent blocks into a single CIST, similar to how overlapping CISTs are handled.

However, I did not see this behavior in STAMP and therefore my CIST prediction and serialization does not address this case.

3.2.2.4. Other details

A few other details remain. First, because multiple threads may concurrently execute CIST exceptions, the exception handlers are synchronized in software to ensure proper serialization. Second, because both going past non-delinquent block conflicts and CISTs create acyclic dependencies among threads (i.e., a thread enters a CIST before an exiting thread commits), threads must commit in dependence order. Order-capture (Section 3.3) ensures correct commit order. Third, in the few cases that a transaction commits before reaching a CIST's CISTend, the commit raises the CIST-Exit exception, releasing any waiting transaction. Fourth, by going past conflicts, transactions may read speculative data and incur faults (e.g., address fault) [37]. I delay handling of such faults until the transaction's predecessors have committed, like DATM [37]. Fifth, non-transactional accesses do not access the CPT and therefore, ignore delinquency. Finally, while WnG does not pre-empt threads waiting at CISTs, pre-emption could be added, similar to lock libraries or always-wait with pre-emption [29].

3.2.2.5. Other Optimizations

I propose two optimizations: (1) to address read-only delinquent blocks and (2) to address unstable delinquent block sets.

First, I address delinquent blocks which are only read by some transactions but read and written by others. To increase concurrency, I wish to serialize read-write and read-only CISTs with respect to each other while allowing multiple concurrent read-only CISTs. To this end, a read-only saturating counter is added to each CPT entry (Figure 3.2); the counter is incremented at commit if a transaction only read the delinquent block and is decremented immediately upon a write. To serialize, while waiting read-write CISTs proceed one by one, all waiting read-only CISTs, irrespective of queue position, proceed when any read-only thread enters the CIST. Because read-only nature is specific

to each transaction, WnG does not gossip read-only information (Section 3.2.1). The second optimization targets transactions with unstable delinquent block sets, where soon after a block is learned to be delinquent, the block ceases to be accessed and is unlearned (Section 3.2.1). In this case, serializing CISTs is ineffective and WnG's default of always-go for conflicts on non-delinquent blocks (Section 3.2.2.1) causes many late aborts. I detect such instability by monitoring the per-commit number of prediction counter decrements for any delinquent block. After a few commits, if this number exceeds an instability-threshold, the unstable transaction's future instances (i.e., same TxID) irreversibly switch to always-wait and gossip the switch to same-TxID transactions in other threads (Section 3.2.1). Transactions with other TxIDs continue to use the WnG policy.

3.3. HTM Mechanisms for Wait-n-Go

Wait-n-GoTM (WnGTM) allows transactions to proceed past conflicts after waiting appropriately (i.e., transactions execute concurrently outside CISTs). Therefore, WnGTM must adopt a multi-reader, multi-writer paradigm to ensure serializability among coexisting readers and writers. Accordingly, I generalize previous TMs' multi-reader single-writer invariant and unordered transactions, to a multi-reader multi-writer paradigm with ordered transactions like DATM. To this end, WnGTM adds three extensions proposed in [42] to previous TMs: a transactional conflict-state to detect conflicts (Section 3.3.2), order-capture to identify and track dependencies (Section 3.3.3), and a timestamp scheme adapted from Timetraveler [39] to detect cycles via both data dependencies and waiting (Section 3.3.4). While the WnG policy can be applied to any STM or HTM, I describe one implementation here. To illustrate that WnG need not alter the coherence protocol; I select TokenTM, which does not change coherence, as my baseline. Previous schemes change coherence for waiting (e.g., LogTM-SE [28]) and conflict detection (e.g., DATM), and degrade scalability by broadcasting dependencies (e.g., DATM). In contrast, these extensions build on TokenTM to avoid coherence changes and on Timetraveler to avoid broadcasts.

3.3.1. TokeTM: Background

TokenTM avoids coherence changes by allowing transactional state to move with blocks upon conflicts, piggybacked on coherence messages. TokenTM resolves conflicts via L1-resident blocks in hardware and resorts to software all-log walks for the rare conflicts via L1-evicted blocks where hardware cannot identify all the conflicting transactions. TokenTM denotes transactionally accessed blocks using per-block read (R) and write (W) bits in the private L1s and, to ensure evicted blocks' transactional state is not lost, pushes a summary of this transactional state to all memory levels. To commit in TokenTM, transactions that have conflicted or evicted blocks release transactional state in software (slow commit), while transactions with all L1-resident blocks release transactional state in hardware via a flash clear of the L1 R and W bits (fast commit). To facilitate these software slow commits and also aborts, TokenTM logs all transactional accesses and old write values. Upon a conflict via an L1-evicted block, the identity of conflictors may not be known (i.e., TokenTM's state in the L2/memory contains only a summary and not the conflictors' identities). In these cases, TokenTM triggers an all-log walk (of all live transactions) to determine the conflictors' identities.

3.3.2. Conflict State: Detecting conflicts in WnGTM

Following TokenTM, WnGTM (1) avoids coherence changes and resolves conflicts in the same manner; (2) WnGTM also pushes its transactional state to all memory levels, as I describe in this section; and (3) WnGTM adopts both fast and slow commits as well as all-log walks to manage transactional state. To detect conflicts under the multi-reader, multi-writer paradigm, WnGTM employs a new transactional (not coherence) state, conflict-state [42], which avoids coherence changes by moving with the blocks upon a conflicting access. The conflict-state consists of a per-block conflict bit, C, signifying that any future access will conflict (i.e., at least one transactional writer exists), and a count of the block's current readers and writers. Non-conflicting reads and writes yield non-conflicted (C=0) and transactionally clean (C=0, count=0) blocks, respectively.

3.3.3. Order-capture: Ordering dependencies in WnGTM

As explained in Section 3.2.2.4, going past conflicts requires dependent transactions to commit and abort in the data dependence order for correct serializability. That is, a transaction may commit only after all its predecessors have committed and must abort if a predecessor aborts (cascaded abort) but must do so only after all the transaction's successors have aborted. Ordering with WnGTM's timestamps would serialize all transactions, not just dependent ones, strictly based on numeric values. To limit serialization, [42] propose order-capture which scalably and explicitly tracks data dependencies so that commits and aborts wait only for predecessors and successors, respectively. To track dependencies, conflict block accessors (i.e., C bit is set) log the predecessor transactions' identities (thread identifier and per-thread dynamic transaction count mentioned in Section 3.2.1 and shown in Figure 3.2); the accessing transaction is the successor because other transactions' accesses occurred earlier. Because predecessors may not be notified upon conflicts (e.g., the block is not in the L1), only the successor logs. In the simple, common case of conflicts on L1-resident blocks, hardware provides the predecessors' identities (piggybacked on coherence messages). A transaction that has proceeded past a conflict slow-commits in software via an exception (like TokenTM), during which the transaction uses its log to wait for its predecessors to commit. Conflict-free transactions do not wait at commit. Analogously, an aborting transaction waits in its abort handler for all its successors to abort. However, because transactions only log predecessors, successors are not known. Consequently, aborting transactions trigger all-log walks to check all transactions' predecessor lists and abort those transactions for which the aborting transaction is a predecessor.

3.3.4. TimeStamps: Detecting cycles in WnGTM

Both waiting and proceeding past conflicts induce inter-transaction dependencies so WnGTM requires cycle detection to ensure serializability. To this end, Voskuilen at al in [42] employ timestamps, as mentioned previously in Section 3.2.1 and Section 3.2.2. Prior schemes detect cycles using static timestamps which are assigned at transaction start [22]. Such static timestamps cause false cycles among independent transactions due to coincidental numerical ordering among the timestamps. To minimize false cycles,

WnGTM adopts Timetraveler's post-dated timestamps [39] which are adjusted dynamically to allow transactions' timestamps to slide flexibly and independently with respect to each other. As any conflict (delinquent or not) may close a cycle and as conflicts occur fairly frequently, Voskuilen at al use hardware to generate timestamps and detect cycles in the simple, common case of conflicts on L1-resident blocks. WnGTM resorts to software to detect cycles in the rare and more difficult case of L1- evicted block conflicts. Also, as CIST waiting already occurs in software, WnGTM detects waiting cycles in software. Crucially, cycle detection is independent of the WnG policy, ensuring that mispredictions cannot corrupt the timestamps.

3.3.5. WnGTM-wait: A simpler variant of WnGTM

The WnG policy allows a successor transaction to proceed concurrently with a predecessor transaction once the predecessor exits the CIST. While it increases concurrency, WnG necessitates the complexity of tracking the predecessor-successor data dependence via order-capture and of enforcing the dependence order to ensure serializability via all-log walks and the log-walk state. To avoid some of this complexity, I consider a variant policy, called WnGwait, where the successor simply waits until the predecessor commits or aborts. That is, CISTs are still serialized but the serialization is elongated until the transaction end instead of the CISTend. In this variant, because the successor waits at the CIST entry before accessing a delinquent block, there is no predecessor-successor data dependence and the associated complexity of order capture and the log-walk state. However, WnGTM-wait still has to detect (1) CISTs via CIST prediction, (2) conflicts via the conflict state, and (3) waiting cycles via timestamps, like WnGTM. I note that WnG-wait is different from always-wait because WnG-wait avoids most CIST-induced aborts by waiting at the CIST entry before accessing the delinquent block whereas always-wait waits upon a conflict usually after multiple accesses to the delinquent block making cyclic waiting, and hence aborts, inevitable. Finally, WnGTM-wait's lower complexity than WnGTM comes at the cost of (1) slower CIST learning due to the extra waiting and (2) lower predecessor-successor concurrency. I evaluate WnGTM-wait's complexity-performance trade-off in my results.

3.4. Related Work

WnG's CIST prediction is inspired by memory dependence prediction [35]. WnG generalizes the centralized scheme for sequential programs in [35] to a distributed scheme for parallel, TM programs. Current HTMs (as discussed in Section 3.1) force conflicting transactions to wait for the others to commit or abort (always-wait), abort the transactions (always-abort), or allow the transactions to proceed, forcing commits and aborts in the conflict induced dependence order (always-go).

LogTM [22], LogTM-SE [28], and UFO [5] use always-wait with logical timestamps to detect deadlocks and trigger aborts. Bobba et al. in [8] reduce (cyclic) waiting in always-wait by prioritizing older transactions over younger ones. The always-wait HTMs wait in hardware via nacks, requiring changes to coherence. While these HTMs do not pre-empt waiting threads, Blake et al. [29] propose a variant of always-wait which serializes entire repeatedly-conflicting transactions by swapping-out some of the conflicting threads until the remaining thread commits and swapping-in other threads. However, the other threads also run into conflicts which causes excessive serialization of entire transactions.

Many HTMs [3,6,7,12,18,27,36] use always-abort. UTM [3], PTM [12], and TokenTM [7] use age priority to abort younger transactions upon conflicts. Other TMs use the amount of work done, instead of age, to prioritize conflicting transactions (e.g., [6,18,36]). TokenTM employs always-abort to avoid coherence changes. OneTM [6] and MetaTM [18] can be configured to use always-wait or always-abort. SigTM [10] employs always-abort but switches to always-wait if a transaction aborts repeatedly.

As discussed before, DATM uses always-go to increase concurrency and uses coherence changes and broadcasts for dependence tracking and cycle detection. Recently, RETCON [30] uses always-go but employs slice re-execution to repair incorrect memory state instead of imposing dependence order on commits and aborts. Because such repair is hard in the general case, RETCON is limited to simple cases, such as counter increments. RETCON shows performance improvements only for the STAMP benchmarks reconfigured to include simple counters (e.g., hash table occupancy counters) and no improvements with the normal configuration. Lazy conflict resolution

(e.g., TCC [16]), a special case of always-go, postpones conflict detection until commit and then attempts to serialize conflicting transactions. While TCC avoids some acyclic aborts, it incurs late, cyclic aborts. Further, lazy conflict resolution requires lazy version management and therefore, hardware write buffers (eager versioning is not possible). These buffers incur complexity and performance degradation when transactions exceed the buffers' finite size. Finally, to correctly detect conflicts at commit and atomically update the memory system with committed write values, transactions' commits are serialized.

In summary, always-wait waits upon conflicts until transactions end — well past the CISTs — incurring long delays, whereas WnG serializes only the CISTs. While always-abort incurs frequent aborts and always-go inevitably incurs late aborts due to cyclic dependencies, WnG avoids many aborts by serializing the CISTs. Unlike RETCON, WnG can handle CISTs comprising arbitrary computation.

Finally, while WnG prediction specifically targets cyclic conflicts, there has been some work on general prediction of conflicts. Blake et al. [29], mentioned above, learn repeated conflicts to avoid scheduling conflicting transactions in parallel. Waliullah et al. [40] learn repeated conflicts and take checkpoints during transactions when a potentially conflicting block is accessed. On aborts, the transaction uses the checkpoints to reduce the extent of rollback. While this scheme reduces lost work upon an abort, WnG avoids aborts, thereby reducing not only lost work but also roll-back, backoff, and waiting.

Table 3.1Hardware parameters

Cores	16, in-order cores
Private L1	32K, 4-way, 64 byte blocks, 1-cycle latency
Shared L2	8M, 8-way, 64 byte blocks, 34-cycle latency
Memory	8GB, 448-cycle latency
Coherence	Directory MESI with full bit-vector sharer list
TokenTM	State bits per block (5 in L1 and 2 in L2) + 14-bit thread id/count per L1 & L2 block
Wait-n-GoTM	C, LW bits per L1 & L2 block (+ TokenTM's 14-bit id/count), 32-entry CPT, instability-threshold = 1.6 (section 2.2.5)

3.5. Methodology

I implement WnGTM by extending TokenTM in the Wisconsin GEMS HTM simulator [21] which uses Simics [20] for full-system simulations. I simulate a SPARC-based multicore running Solaris 10. Table 3.1 summarizes the simulated system's parameters. Using GEMS's user-level exception handlers, I simulate CIST-Enter and CIST-Exit exceptions and order-capture's all-log walks, as well as TokenTM's slow commits and aborts. I evaluate WnGTM using STAMP [10], as shown in Table 3.2. I select small or medium input datasets based on simulation feasibility. The table also shows the fraction of TokenTM's execution time spent in transactions (%tx time), the transaction length described quantitatively as the number of instructions per transaction (#instrs/tx) and qualitatively (tx length), the amount of contention expressed quantitatively as the number of aborts per commit in TokenTM (#aborts/commit) and qualitatively (contention), and the base-case performance shown as 16-core TokenTM's speedups over sequential runs (TokenTM vs. Single). Labyrinth's speedup does not scale beyond 8 cores without early-release [9]. The data mostly match those in [9] and show that STAMP covers a wide spectrum of transactional behavior. To account for statistical variations, I use enough randomly-perturbed runs to achieve 95% confidence [2].

3.6. Experimental Results

I first compare the conflict resolution policies, always-abort, always-wait, and always-go, against the WnG policy. To explain the performance differences I provide an execution time breakdown. Next, I show CIST and delinquent block statistics. I then isolate the performance impact of my various optimizations for WnG. Finally, I show sensitivity to the CPT size.

3.6.1. Performance

Table 3.4 describes CISTs using the number of all and read-only delinquent blocks per benchmark (#delinquent blks and #rd-only delinq blks), the fraction of dynamic transactions that contain CISTs (%tx with CISTs), read-only CISTs (%tx with rd-only CISTs), and overlapping CISTs (%tx with overlap CISTs), the average number of all and overlapping CISTs per transaction containing CISTs (#CISTs/tx and #overlap CISTs/tx), the average number of CISTs per set of overlapping CISTs (#CISTs/overlap set), and the average predicted and real CIST lengths in number of instructions (predicted/real CIST length). I omit ssca2, vacation-lo, and vacation-hi as they do not have delinquent blocks. Most higher-contention benchmarks have only a few delinquent blocks (Table 3.4, column 1), a small subset of which are read-only (column 2). Nevertheless, a substantial fraction of the transactions contain CISTs (column 3), correlating with the high number of aborts (Table 3) and highlighting the need to serialize CISTs. Genome is the only exception where many aborts are due to non-delinquent blocks. Intruder and labyrinth have considerable fractions of transactions containing read-only CISTs (Table 3.4, column 4), making the read-only optimization (Section 3.2.2.5) important for these benchmarks. Over a third of intruder's and labyrinth's transactions contain overlapping CISTs (Table 3.4, column 5) so handling overlapping CISTs helps these benchmarks to some extent. Although many transactions contain CISTs, individual transactions contain fewer than two CISTs (Table 3.4, column 6), except for labyrinth whose long transactions contain many (mostly overlapping) CISTs (Table 3.4, columns 7, 8). Because transactions have few CISTs, the likelihood of multiple CISTs forming cyclic dependencies is small. Predicted and real CIST lengths (Table 3.4, column 9) are similar, indicating the CISTend prediction is accurate. Intruder and labyrinth are exceptions and

the difference is almost fully due to read-only CISTs (not shown). Finally, because the CISTs are much shorter than the transactions (Table 3.2, column 2 vs. Table 3.4, column 9), WnG exploits concurrency outside CISTs, unlike always-wait. At the same time, transactions and CISTs comprise many instructions, making special-case optimizations like RETCON unlikely to be effective broadly. Though I show only the average CIST lengths and the individual CIST lengths vary, the above two points hold for individual CISTs as well.

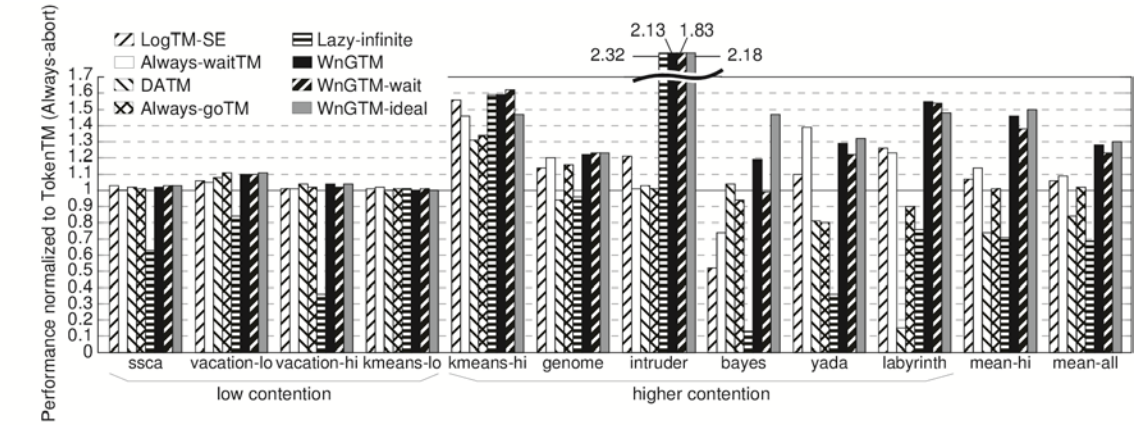

Fig 3.8 Performance

For a fair comparison, I use good configurations for each HTM, as discussed in Section 3.4: age-based priority for TokenTM; 2K-entry Bloom filters and age-based priority for LogTM-SE; perfect broadcasts, perfect cycle detection, and perfect aborts for DATM; and 2K-entry Bloom filters and infinite write buffers for lazy-infinite. For WnGTM, I use (1) CIST-Enter and CIST-Exit exceptions for waiting, (2) order-capture to identify dependencies (including all-log walks upon L1-evicted block conflicts), and (3) post-dated timestamps with the potential for false cycles.

Figure 3.7 shows the speedups over TokenTM (Y axis) for LogTM-SE, Always-waitTM, DATM, Always-goTM, lazy-infinite, WnGTM, WnGTM-wait, and WnGTM-ideal (X axis). The X axis groups the low- and higher-contention benchmarks (the medium- and high-contention benchmarks in Table 3.2, column 5), and shows the

averages for the higher-contention benchmarks and for all benchmarks, denoted as mean-hi and mean-all, respectively. Table 3.3 gives the number of aborts per commit.

Table 3.2 Benchmarks

	1	2	3	4	5	6
Benchmark (input)	%tx time	#instrs/tx	tx length	#aborts/ commit	contention	TokenTM vs. Single
ssca2(m)	8	22	short	~0	low	6.2
vacation-lo(m)	95	2015	med	~0	low	13.3
vacation-hi(m)	96	2730	med	~0	low	13.9
k-means-lo(m)	4	155	short	0.1	low	7.3
k-means-hi(m)	44	155	short	0.5	med	7.6
genome(m)	44	1237	med	0.4	med	3.5
intruder(m)	95	212	short	3.2	high	1.7
bayes(s)	94	41978	long	4.0	high	2.7
yada(s)	98	6263	long	1.7	high	3.7
labyrinth(s)	99	249450	long	10.4	high	0.6

Figure 3.7 shows that most conflict resolution policies perform similarly for the low-contention benchmarks but differ significantly for the higher-contention benchmarks. LogTM-SE incurs generally fewer aborts than TokenTM (Table 3.3, columns 1, 2) but on average, performs similarly due to significant waiting. By avoiding Bloom filters' false conflicts and static timestamps' false cycles, Always-waitTM aborts less often than LogTM-SE in intruder and bayes (Table 3.3, columns 2, 3). While Always-waitTM's fewer aborts in bayes translate to better performance, Always-waitTM's exception

overhead for waiting negates the abort advantage in intruder (LogTM-SE waits via coherence nacks). As expected, the always-go HTMs (DATM, Always-goTM, and lazy-infinite) incur fewer aborts than TokenTM (Table 3.3, columns 1, 4, 5, 6) by allowing acyclic dependencies but incur numerous CIST-induced late aborts. Labyrinth performs much worse with DATM due to especially late aborts. Lazy-infinite performs worse than TokenTM in some cases (e.g., vacation-hi and yada) and better in others (e.g., k-means-hi and intruder). Recall that lazy-infinite uses an infinitely-large write buffer and therefore does not incur any eviction-related overheads such as exceptions and log walks while the other schemes include these overheads. Therefore, lazy-infinite performance is an upper bound and should be not be used to compare directly against the other schemes. Vacation's higher abort rate stems from its moderate number of acyclic conflicts. Lazy-infinite's serialized commits extend the time that a transaction is active in vacation, increasing the likelihood of acyclic conflicts and aborts. Lazy-infinite's late, cyclic aborts significantly hurt bayes and yada. In contrast, WnGTM avoids acyclic aborts and many cyclic aborts in TokenTM to achieve better performance (Table 3.3, columns 1, 7). WnGTM-wait's extra waiting (Section 3.3.5) reduces aborts over WnGTM in all the higher-contention benchmarks except intruder where WnGTM-wait's slower learning results in more aborts (Table 3.3, columns 7, 8).

Table 3.3 Aborts per commit

	1	2	3	4	5	6	7	8	9
Benchmark	TokenTM	LogTM-SE	Always-waitTM	DATM	Always-goTM	Lazy-infinite	WnGTM	WngTM-wait	WnGTM-ideal
ssca2	~0	~0	~0	~0	~0	~0	~0	~0	~0
vacation-lo	~0	~0	~0	~0	0	2.5	0	0	0
vacation-hi	~0	~0	~0	~0	~0	2.6	~0	~0	~0
k-means-lo	0.1	~0	~0	0.1	~0	0.1	~0	~0	~0
k-means-hi	0.5	0.1	0.2	0.4	0.4	0.3	0.1	~0	0.1
genome	0.4	0.2	~0	0.2	0.1	0.2	~0	~0	~0
intruder	3.2	3.7	3.0	1.4	1.5	0.6	0.5	0.6	0.4
bayes	4.0	3.9	2.9	2.1	2.1	3.0	1.2	0.7	1.2
yada	1.7	0.7	0.6	1.1	1.1	1.1	0.4	0.3	0.5
labyrinth	10.4	8.8	8.7	22.4	8.9	8.0	2.1	1.5	2.2

On average, the always-wait HTMs, LogTM-SE and Always-waitTM, perform better than TokenTM with Always-waitTM achieving 14% and 9% improvements for the higher-contention benchmarks and all the benchmarks, respectively (mean-hi and mean-all in Figure 3.7). Among the always-go HTMs, Always-goTM performs similar to, whereas DATM and lazy-infinite perform worse than, TokenTM. WnGTM achieves significant improvements of 46% (mean-hi) and 28% (mean-all) over TokenTM,

respectively. WnGTM's high speedup for intruder comes from incurring far fewer aborts than TokenTM (Table 3.3, columns 1, 7). WnGTM lags behind only Always-waitTM for yada — due to yada's unstable delinquent block set — and lazy-infinite for intruder — because WnGTM incurs some abort overhead whereas lazy-infinite's aborts are overhead-free; however, the lag is under 9% in both cases. By serializing the CISTs, WnGTM drastically reduces the abort rate (Table 3.3, column 7) and performs significantly better than the previous policies, including Always-wait which is the best previous policy. Thus, WnGTM achieves its goal of increasing concurrency while avoiding many cyclic aborts. By trading-off concurrency for complexity, WnGTM-wait performs worse than WnGTM, with significant degradation in intruder, bayes, and yada. Thus, WnGTM-wait sacrifices some performance to eliminate some complexity. Nevertheless, WnGTM-wait performs 39% better than TokenTM for the higher contention benchmarks, illustrating the importance of avoiding cyclic aborts via CIST-prediction. WnGTM-wait is a viable option for designers who wish to reap the benefits of CIST prediction while avoiding the complexity of order-capture and the log-walk state. WnGTM-ideal is only 2% better than WnGTM and has a similar abort rate (Table 3.3, columns 7, 9), indicating that the performance overhead from CIST exceptions, all-log walks, and timestamp-induced false cycles is small. In two cases, k-means-hi and labyrinth, WnGTM-ideal performs slightly worse than WnGTM. In k-means-hi, WnGTM's exception overhead spaces conflicting transactions apart in time, so that the transactions are more likely to reach commit in their dependence order and thereby avoid waiting at commit. This spacing is absent in WnGTM-ideal causing extra waiting. In labyrinth, WnGTM-ideal's infinite CPT serializes an excessive number of CISTs, increasing waiting. Both programs' waiting is due to program dependencies which cannot be removed in WnGTM-ideal.

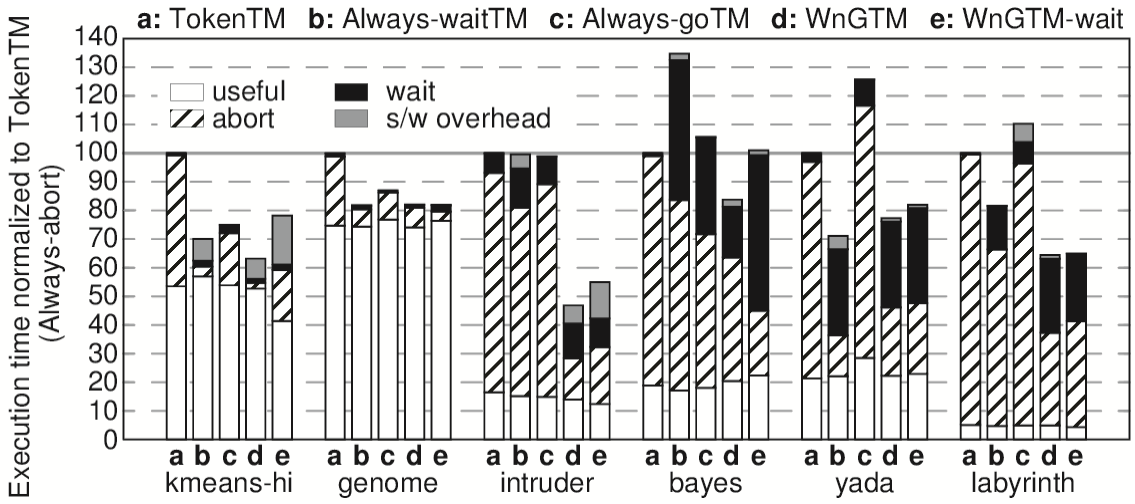

Fig 3.9 Execution time breakdown

While WnGTM avoids acyclic aborts and many cyclic aborts, Table 3.3 (column 7) shows that WnGTM still experiences some aborts in the higher contention benchmarks. In intruder, these aborts stem primarily from delinquent read-only blocks transitioning to read-write and from learning-related aborts. Learning related aborts include aborts before all conflicting transactions have learned a block to be delinquent and aborts from accessing a delinquent block beyond its current CISTend (Section 3.2.1). Bayes' aborts are also primarily learning-related. Yada's unstable delinquent block set leads to many cyclic aborts through non-delinquent blocks. Yada additionally incurs aborts due to learning and to rushing, where a transaction that is a predecessor to another transaction for one CIST reaches a later CIST after its successor has entered that later CIST. Finally, labyrinth incurs learning-related and rushing-induced aborts.

To explain WnGTM's performance, Figure 3.8 shows an execution time breakdown for TokenTM (representing always-abort), Always-waitTM, Always-goTM, WnGTM, and WnGTM-wait, normalized to TokenTM. I break total time into useful (includes non-transactional work), abort (includes lost transactional work, futile waiting, roll back of log, and back-off), wait (includes fruitful waiting and waiting to commit or abort), and software overhead (CIST exceptions, slow commits, and log walks). I show only the higher-contention benchmarks where conflict resolution matters. From Figure 3.8, I see

that TokenTM (always-abort) loses a significant fraction of time to aborts, mostly due to back-off. Although Always-waitTM and Always-goTM abort less than TokenTM (Table 3.3, columns 1, 3, and 5), the policies lose a significant amount of time to late, cyclic aborts, which lead to futile waiting (Always-waitTM) and lost work (Always-goTM). WnGTM loses a smaller amount of time to aborts compared to TokenTM, Always-waitTM, and Always-goTM, which correlates with WnGTM's fewer aborts (Table 3.3, columns 1, 3, 5, and 7). Still, WnGTM loses a moderate amount of time to aborts, for the reasons stated above. Further, WnGTM waits mostly less than Always-waitTM because WnGTM waits only for conflicting delinquent-block accesses and only until the CISTend, whereas Always-waitTM waits for all conflicting accesses and until the transaction end. Moreover, Figure 3.8 confirms my previous claim that WnGTM's software overhead, including CIST exceptions, is small. WnGTM-wait incurs more aborts than WnGTM due to slower learning (Table 3.3, columns 7, 8) and hence loses more time to aborts in intruder. In bayes and yada, WnGTM-wait's waiting until the transaction end instead of the CISTend results in longer waiting.

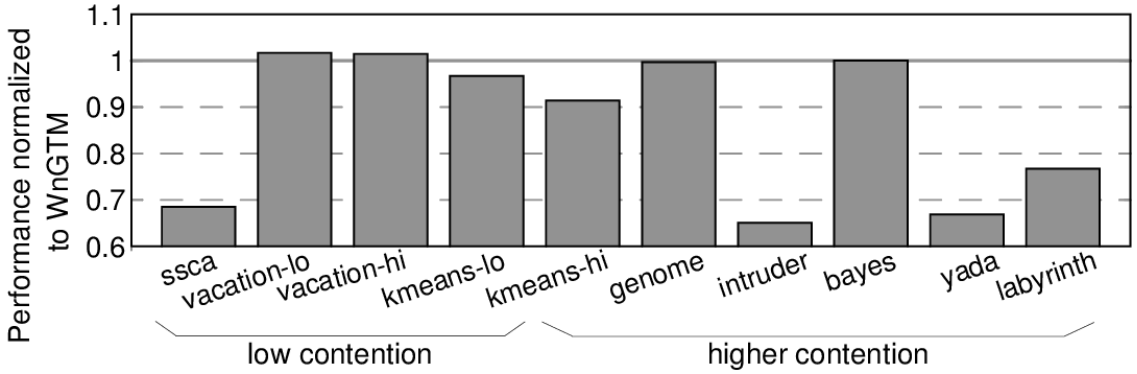

Fig 3.10 WnGTM versus wait-pre-empt

Finally, I compare WnGTM to always-wait with thread preemption, called wait-pre-empt, which learns repeated conflicts in software to pro-actively serialize entire transactions at their start [29]. Wait-pre-empt switches out long transactions or stalls short transactions that are predicted to conflict. Because changing the thread scheduler's preemption policy is involved, I simulate the effect in hardware: an 8-core wait-pre-empt running 16 threads executes on 16 cores and stalls/unstalls cores so that exactly 8 cores

are unstalled at a time. While normal thread switching incurs software and cold-cache overhead, my thread switching is zero cost, favoring wait-preempt. Figure 3.9 shows wait-pre-empts' performance normalized to WnGTM's for 8 cores. Because the switched-in threads also run into conflicts, causing excessive serialization of entire transactions, wait-pre-empt performs worse than or at par with WnGTM for all the benchmarks. Though ssca2 has few conflicts, it has many transactions and the per-transaction overhead of checking the prediction in software worsens performance.

Table 3.4 CIST statistics

	1	2	3	4	5	6	7	8	9
Benchmark	#delinquent blks	#rd-only delinquent blks	%tx with CISTs	%tx with rd-only CISTs	%tx with overlap CISTs	#CISTs/tx	#overlap CISTs/tx	#CISTs/overlap set	Predicted/read CIST length
k-m-lo	10	0	25	0	0	1	0	-	2/1.2
k-m-hi	16	0	27	0	0	1	0	-	6/5
Genome	11	1	1	~0	0	1	~0	-	153/92
Intruder	240	22	77	60	35	1.5	0.5	2	43/12
Bayes	23	8	64	3	6	1.2	0.1	2.3	3.5K/2.9K
Yada	85	4	34	~0	2	1	~0	2.3	382/157
Labyrinth	41	40	84	32	41	11	0.5	20	11.4K/2.2K

3.6.2. CIST statistics

Table 3.4 describes CISTs using the number of all and read-only delinquent blocks per benchmark (#delinquent blks and #rd-only delinq blks), the fraction of dynamic transactions that contain CISTs (%tx with CISTs), read-only CISTs (%tx with rd-only CISTs), and overlapping CISTs (%tx with overlap CISTs), the average number of all and

overlapping CISTs per transaction containing CISTs (#CISTs/tx and #overlap CISTs/tx), the average number of CISTs per set of overlapping CISTs (#CISTs/overlap set), and the average predicted and real CIST lengths in number of instructions (predicted/real CIST length). I omit ssca2, vacation-lo, and vacation-hi as they do not have delinquent blocks. Most higher-contention benchmarks have only a few delinquent blocks (Table 3.4, column 1), a small subset of which are read-only (column 2). Nevertheless, a substantial fraction of the transactions contain CISTs (column 3), correlating with the high number of aborts (Table 3) and highlighting the need to serialize CISTs. Genome is the only exception where many aborts are due to non-delinquent blocks. Intruder and labyrinth have considerable fractions of transactions containing read-only CISTs (Table 3.4, column 4), making the read-only optimization (Section 3.2.2.5) important for these benchmarks. Over a third of intruder's and labyrinth's transactions contain overlapping CISTs (Table 3.4, column 5) so handling overlapping CISTs helps these benchmarks to some extent. Although many transactions contain CISTs, individual transactions contain fewer than two CISTs (Table 3.4, column 6), except for labyrinth whose long transactions contain many (mostly overlapping) CISTs (Table 3.4, columns 7, 8). Because transactions have few CISTs, the likelihood of multiple CISTs forming cyclic dependencies is small. Predicted and real CIST lengths (Table 3.4, column 9) are similar, indicating the CISTend prediction is accurate. Intruder and labyrinth are exceptions and the difference is almost fully due to read-only CISTs (not shown). Finally, because the CISTs are much shorter than the transactions (Table 3.2, column 2 vs. Table 3.4, column 9), WnG exploits concurrency outside CISTs, unlike always-wait. At the same time, transactions and CISTs comprise many instructions, making special-case optimizations like RETCON unlikely to be effective broadly. Though I show only the average CIST lengths and the individual CIST lengths vary, the above two points hold for individual CISTs as well.

3.6.3. Impact of optimizations

In Figure 3.10 I isolate the impact of my optimizations for handling overlapping CISTs (Section 3.2.2.2), handling read-only CISTs (Section 3.2.2.5), gossiping CIST information (Section 3.2.1), and handling unstable delinquent-block sets (Section

3.2.2.5). The Y axis shows WnGTM's performance as I add each of these optimizations, beginning with WnGTM without optimizations (none) and ending with the complete WnGTM (unstable delinquent set). Performance is normalized to that of the complete WnGTM. The X axis shows the benchmarks (only higher-contention). Each optimization impacts a few benchmarks. While both intruder and labyrinth have many overlapping CISTs (Table 3.4, column 5), labyrinth responds well to my overlap optimization whereas intruder does not because it has only two CISTs per overlapping set (Table 3.4, column 8). Intruder and labyrinth have many read-only CISTs (Table 3.4, column 4), and gain with the read-only optimization. Gossip accelerates CIST learning, aiding both kmeans-hi and labyrinth. Finally, yada—and, to a lesser extent, bayes—have unstable delinquent-block sets and improve with the corresponding optimization.

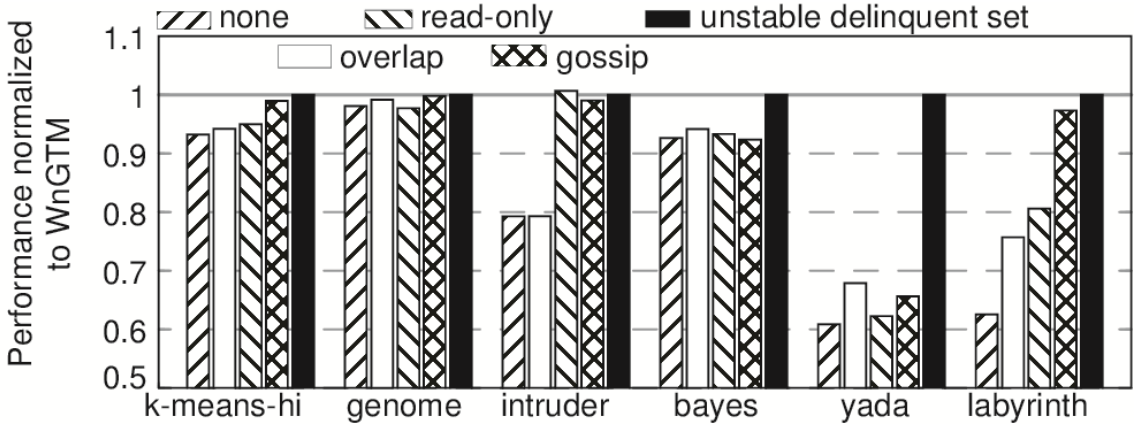

Fig 3.11 Optimizations

3.6.4. Sensitivity to CPT

Figure 3.11 shows WnGTM's sensitivity to the CPT size for the higher-contention benchmarks. The Y axis shows WnGTM's performance with the CPT size varying as 8, 16, 32, 64, and 128 entries, normalized to that of WnGTM with a 32-entry CPT (default). WnGTM's performance improves with larger CPTs for intruder, bayes, and yada which have many delinquent blocks (Table 3.4, column 1). Labyrinth also has many delinquent blocks, but accesses only a few at a time, so does not benefit from larger CPTs. Overall, even 16 entries would work reasonably well, and little benefit is seen beyond 32 entries.

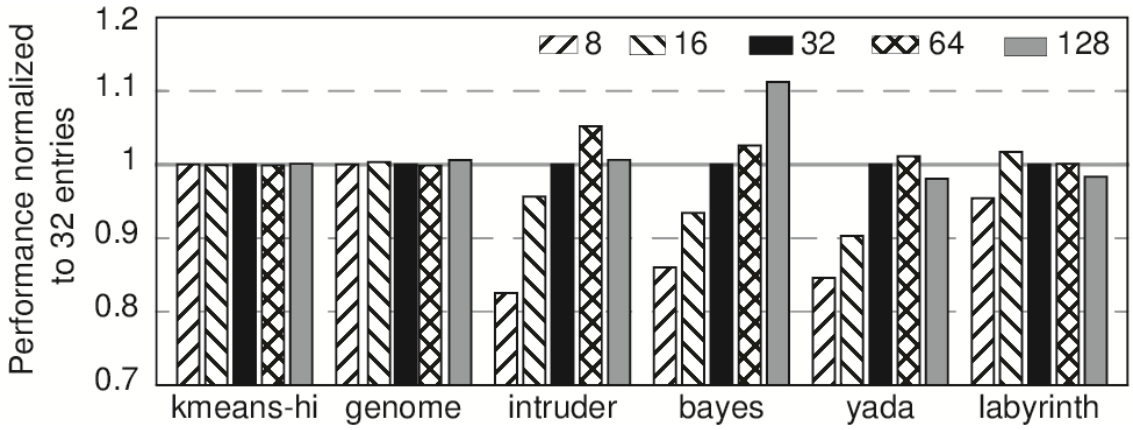

Fig 3.12 CPT size sensitivity

3.7. Conclusion

Conflict resolution policies significantly impact TM performance. While previous policies, always-abort, always-wait, and always-go have their advantages, they degrade performance in the presence of contention by limiting concurrency (always-abort, always-wait) or by incurring late aborts (always-go). Further, none of the policies avoid aborts due to cyclic dependencies. I stipulated that conflict resolution policies should increase concurrency while avoiding cyclic aborts. To that end, I proposed Wait-n-GoTM (WnGTM) based on the key observation that most cyclic dependencies (and hence aborts) are caused by threads interleaving accesses to a few heavily-read-write-shared delinquent data cache blocks. I referred to the section of code containing all accesses to a particular delinquent block as a cycle inducer section (CIST). The Wait-n-Go (WnG) policy employs a CIST predictor in hardware to predict when transactions need to be serialized (i.e., upon entering a CIST) and for how long to avoid many cyclic aborts (i.e., until exiting a CIST). To implement WnG, I along with my colleague Gwendolyn Voskuilen proposed WnGTM in [42] which adds three mechanisms to previous HTMs: (1) a new transactional state, conflict-state, to allow multiple readers and writers; (2) order-capture, to scalably identify data dependencies among transactions; and (3) an adaptation of a hardware timestamp scheme to detect cycles. To keep the complexity manageable, WnGTM handles in hardware only the simple cases in both the policy and mechanisms (e.g., updating the predictor and handling L1-resident blocks) and pushes the

more-difficult cases to software (e.g., serializing CISTs and handling L1-evicted blocks). Using 16-core simulations of STAMP, I showed that WnGTM achieves, on average, 46% and 28% speedups over always-abort (TokenTM) for the higher-contention benchmarks and all the benchmarks, respectively, with low-contention benchmarks remaining unchanged. In comparison, always-go (DATM) and always-wait (LogTM-SE), perform worse than and 6% better than TokenTM, respectively. To combat the programmability issues with locks, HTM support has begun to appear in commercial products and many STMs are being developed. Because WnG is applicable to HTMs and STMs, WnG is likely to be important for improving TM performance and performance robustness.

4. ADAPTIVE FLOW CONTROL

4.1. Introduction

As the microprocessor industry moves towards 16+ cores per chip, the adoption of multi-hop networks as the interconnection fabric is inevitable because neither buses nor crossbars scale adequately. Ideally, such multi-hop networks must provide (1) low-latency because all L1 cache misses are exposed to network latencies, and (2) high bandwidth to support the larger number of cores.

Traditionally networks have been designed to handle link contention by using input buffering (to stall all but one of the contending flits) and backpressure (to prevent stalled/buffered flits from being overwritten by other incoming flits). Unfortunately, buffers consume a significant part of on-chip network energy (e.g., 30-40%). Circuit techniques such as buffer resizing or fine-grained gating incur implementation severe complexities (see Section 4.3.1). Accordingly, recent work addresses the problem of buffer dynamic energy by employing buffer bypass techniques [43], [44]. Approaches that target both static and dynamic buffer energy by using backpressureless routing and eliminating the use of buffers have also been proposed [45], [46]. Such backpressureless routers handle link contention either by using well-known deflection/hot-potato routing [48] or by dropping packets/flits [46] upon contention. The first variant deflection routing – ensures that all incoming flits leave on some outgoing link, even if it is a misroute (from which the flit will eventually recover). The second variant employs the strategy of dropping all but one of the contending flits instead of misrouting them. Such dropped flits are later retransmitted. At low network loads, backpressureless routing is efficient because link contention, and hence misrouting (or dropping of flits), is rare.

Unfortunately, backpressureless routers suffer from poor performance and energy at higher loads because the misrouting/dropping caused by link contention leads to

increased link utilization, which creates a positive feedback cycle because increased link utilization further increases link contention. Consequently, backpressureless networks saturate at lower throughputs than backpressured networks.

Moscibroda et al., in their case for backpressureless routers, have argued that the network load offered by typical workloads is indeed low [45]. However, their measurements were conducted on multiprogrammed sequential workloads. I show that the network load is not always low for commercial benchmarks running on multi-threaded cores. I make the key observation that load varies significantly across applications and, to a lesser extent, over time and space within the network (e.g., hotspots, program phases). A consequence of this observation is that backpressureless and backpressured networks are not robust in performance-energy across the spectrum of high and low loads – i.e., at high loads backpressureless networks suffer considerable performance and energy disadvantage compared to backpressured networks; and the energy disadvantage reverses at low loads. Performance-energy robustness is important especially for multicores which target general-purpose computing where often applications exert vastly diverse loads on the network.

To address this robustness issue, I along with my colleagues Yu-Ju Hong, Mithuna Thottethodi and T.N. Vijaykumar propose Adaptive Flow Control (AFC) [55] – which dynamically adapts between backpressureless and backpressured flow control to approach the best of both worlds, thereby improving performance-energy robustness. Individual AFC routers dynamically switch between backpressured and backpressureless modes of operation. AFC does not use global, network wide mode switching because a distributed, synchronized operation to ensure that all routers are switched, while applications are running, may be impractical. As such, at any instant of time some routers may be in backpressureless mode and the others in backpressured mode; and, a given router may switch modes over time.

There are two key challenges for AFC, one in the common case of unvarying (high or low) load and the other in corner cases of varying load. In the first case, the routers should be in the appropriate mode of operation to achieve good performance-energy. To avoid per-application tuning of the modes, I propose *local contention thresholds*, my first

novel mechanism, derived statically at design-time based solely on network loading and independent of other application characteristics. Routers where the locally-measured link contention exceeds the thresholds switch to backpressured mode, and vice versa.

Second, AFC must ensure the correctness in the corner cases of flow control interactions between adjacent routers in different modes of operation (e.g., in transient conditions when the load is changing or under high spatial variation in load). Depending on the direction of communication, there is an easy case and a difficult case. In the easy case of a backpressured-mode router communicating with a backpressureless-mode router, no additional safeguards are needed because backpressureless-mode routers are prepared to accept flits on every cycle. However, communication from backpressureless-mode routers to backpressured-mode routers is difficult because backpressured-mode routers, by definition, cannot always accept flits that a backpressureless mode router may send. To address this concern, I propose *gossip-induced mode switches*, my second novel mechanism, wherein backpressureless-mode routers are forced to switch to backpressured-mode because of contention at a neighboring (backpressured) router even though the backpressureless router may not observe local contention. AFC infers contention at the neighboring nodes from locally-visible credit backflows that are used for backpressured flow control.

Finally, because AFC (like backpressureless routing) may route flits of a single packet independently, AFC incurs the area and dynamic energy overhead (compared to a backpressured router) of wider flits requiring wider buffers, crossbars and links to carry per-flit routing information. At low loads, the energy overhead is more than compensated by AFC's ability to eliminate both static and dynamic buffer energy due to its backpressureless mode where all its buffers are power-gated [49]. Such coarse-grained gating does not incur the implementation complexities of fine-grained gating mentioned above. At high loads, where dynamic energy dominates, naively using traditional backpressured mode incurs the full energy overhead. Instead, AFC compensates for the area and energy overhead of the wider flits by leveraging flit-by-flit routing to optimize the backpressured mode. Traditional virtual channel (VC) flow control allocates and releases VC buffers at per-packet granularity to ensure that flits of a packet are always

routed together because individual flits do not contain routing information. In contrast, because a packet's flit may be routed independently by a backpressureless-mode router to a backpressured-mode router, AFC must support flit-by-flit routing even in its backpressured mode. Because the purpose of VCs is to prevent intermingling of flits from multiple packets, flit-by-flit routing simplifies VC allocation. AFC leverages this observation both (1) to improve performance by increasing the number of VCs while shortening the router pipeline via *lazy VC allocation* proposed by Hong et al in [55] at the next router, and (2) to reduce energy by shrinking the per-VC buffer and the total buffer (despite having more VCs).

Neither of these two optimizations is possible in traditional backpressured networks. Such lazy VC allocation is not possible due to basic correctness requirements of VC flow-control. Such buffer reduction is also not possible because conventional VC allocation does not scale to a large number of VCs and because reducing per-VC buffer depth has a significant impact on performance. AFC's shallower buffers recapture a significant fraction of the energy overhead of wider flits and more than compensate for the area overhead of supporting both flow-control mechanisms.

In summary, the chapter's contributions are:

- I demonstrate that backpressured and backpressureless networks are not robust in performance and energy across the spectrum of high and low loads.
- To address this robustness issue, I propose an adaptive flow control (AFC) router which employs *local contention thresholds, gossip-induced mode-switch*, and *lazy VC allocation* proposed by Hont et al. [55]. The first mechanism maximizes performance (and minimizes energy) in the common case, and the second mechanism ensures correctness in corner cases. The third mechanism exploits flit-by-flit routing in AFC's backpressured mode to simplify VC allocation and reduces the buffer requirements by a factor of two in AFC's backpressured mode.
- Simulations using commercial workloads and SPLASH-2 confirm AFC's robustness by showing that AFC achieves performance and energy that are closer to that of the better of backpressured and backpressureless routers.

The rest of the chapter is organized as follows. Section 4.2 analyzes the impact of flow control on performance and energy. Section 4.3 describes adaptive flow control. Section 4.4 describes my evaluation methodology. Section 4.5 discusses experimental results. Related work is described in Section 4.6. Finally, Section 4.7 concludes this chapter.

4.2. Impact of flow control on performance and energy

Backpressured and backpressureless flow controls differ primarily in how they handle link contention. From this difference, a number of other secondary differences emerge. To illustrate the differences, consider how the following scenario is handled by the two flow control techniques: two flits at two different input ports of a router contend for the same output port.

In traditional backpressured networks, one flit is allowed to traverse the desired output link while the other is buffered locally. To prevent subsequent flits from overwriting the stalled flit, backpressure is implemented via credit tokens which let neighboring routers know whether buffers are free. To ensure that the stalled flit does not prevent subsequent flits from utilizing idle links, the input queues employ multi- flit buffers. To ameliorate head-of-line (HOL) blocking in such input-queued routers, routers employ multiple VCs per physical channel. The operation of a canonical backpressured router may be viewed as four key steps (not necessarily corresponding to pipeline stages, as explained later). In the first step, the header flit of a packet is routed (R) to a set of output ports. The second step is the VC allocation (VCA) stage where the header flit requests a VC from among the free VCs on the output ports of interest. In the switch arbitration (SA) step, flits with an allocated VC compete for output ports. In the fourth step the flit traverses the switch (ST) and links (LT) (which may take multiple cycles) to be deposited at the input buffers of the neighboring routers.

The above steps may not correspond to pipeline stages because of several performance/energy optimizations including (a) look-ahead routing (LAR) [50], wherein the routes are computed one hop earlier, (b) speculative overlapping of dependent functions, (e.g., [52]) and (c) aggressive router microarchitectures that can exploit other buffer/crossbar bypass paths to minimize router delay and energy [44]. Table 3.1 shows

an ideal two-stage backpressured router in which I charitably assume that VCA occurs in zero cycles. Realistically, VCA delay can be hidden only by successful speculation, which is more likely at low loads.

Backpressureless flow-control, on the other hand, allows one contending flit to progress on the desired port. The other flit is either dispatched to an output port that may potentially take it farther from its destination (i.e., deflection) or dropped altogether. In either case, the routers are backpressureless since they are always ready to accept new flits because the old flits are either deflected or dropped.

Misrouting vs. Dropping flits: In this paper, I focus on the flit-by-flit deflection routing variant of backpressureless routing [45] because the variant that drops packets saturates at lower loads, even according to the original paper [46]. Because deflection routers ensure that each incoming flit is dispatched on an output port in each cycle [48], no flit is ever blocked. Consequently, deflection routers effectively avoid deadlocks and HOL blocking without the use of VCs.

Key complexities of deflection-based backpressureless routing: Recent critiques of backpressureless routing [46], [52] have focused on two key complexities in backpressureless routing, concluding that they must be overcome before such routing becomes practical. I observe that the complexities are not fundamental and arise because of specific design choices – there exist alternative backpressureless designs that avoid these complexities altogether.

The first perceived complexity is that deflection routers require worst-case buffering at *each* node for reordering and reassembly to handle the possibility of out-of-order flit delivery. Specifically, this complexity may be divided in to two categories: buffering for expected packets (the easy case) and buffering for unexpected packets (the difficult case). In the easy case, reordering and reassembly does not impose any additional cost for expected packets because expected packets are those for which a coherence request has been sent, which implies that there exists a (local) MSHR entry to receive the packet. MSHRs provide such receive side buffering functionality even in traditional backpressured networks where flits from different packets may be intermingled because they may arrive on different physical/virtual channels, as also noted by Moscibroda *et al.*

[45]. One may think that backpressureless will further complicate receive side buffering because flits of the *same* packet may arrive in arbitrary order in backpressureless routers, whereas backpressured networks can only see arbitrary intermingling of flits across different packets. However, there is no additional complexity as both cases require a single random access memory array for MSHR buffers.

Table 4.1Router Pipeline Stages

Flow Control	Router Stage 1	Router Stage 2	Link Traversal
Backpressured	SA (PV → P) LAR in parallel VCA (0-cycles)	ST + partial LT	Partial LT + input BW
Backpressureless	R _ SA (P → P)	ST + partial LT	Partial LT + Latch-write
AFC (backpressureless mode)	Same as backpressureless		
AFC (backpressured mode)	SA (PV → P) LAR in parallel	Same as back-pressured	Same as back-pressured Lazy VCA

The difficult case involves unexpected packets which may occur due to dirty-writebacks. Such unexpected packets do not have pre-allocated MSHR entries to serve as receive-side buffers. In this case, a naive backpressureless implementation will indeed require worst-case buffering at *each* node for as many write-buffer entries in the *entire system*. However, such worst-case buffering can be avoided by using coherence protocol variants that (1) pre-allocate an MSHR entry at the destination, and (2) hold writeback buffer data till such pre-allocation is possible. Note, in the absence of such restrictions even backpressured networks will see an increase in receive-side buffering, albeit less than backpressureless networks.

A second perceived complexity of deflection routing in [46], [52] is that it is fundamentally slow because it requires implementation of hardware priorities to ensure livelock freedom. This complexity is specific to implementations that use hardware priorities in order to guarantee that output ports are assigned to older (higher priority) flits before being assigned to younger (lower priority) flits. Such priorities ensure that the oldest flit at each router is never misrouted, hence guaranteeing forward progress. However, there are alternative implementations that avoid the use of priorities (which are necessary only for deterministic livelock freedom) and instead, rely on randomization and probabilistic guarantee of livelock freedom for the backpressureless router, as done in the Chaos router [47]. I emphasize that the probabilistic nature does not make the guarantee weak because the probability of a flit not reaching its destination diminishes with each hop and can eventually be made arbitrarily small (i.e., smaller than any adversarially-chosen ϵ ($0 \leq \epsilon \leq 1$)).

Summary: For the backpressured and backpressureless implementations described above, I have the following performance energy expectations. On the performance front, backpressureless networks are comparable to backpressured networks at low loads, but are significantly worse at high loads due to excessive misrouting and early saturation. On the energy front, backpressured networks achieve lower energy consumption that backpressureless networks at high loads. However, backpressureless networks achieve lower energy consumption than backpressured networks at low loads, by completely avoiding both static and dynamic buffer energy. The above observations, in conjunction with the fact that there are significant variations in network load across (and to a lesser extent, within) applications, directly make a case for an adaptive flow-control mechanism that captures the best of both worlds.

4.3. Adaptive Flow Control

I describe the operation of AFC in terms of the following three questions. First, Section 4.3.1 answers the question: *What are the mechanisms that enable each AFC router to operate in either backpressured or backpressureless mode?* The next two subsections deal with the policy questions to achieve good performance and energy in the common case of uniform (high or low) load: *When do the* ***forward mode-switch*** *from*

backpressureless mode to backpressured mode and the ***reverse mode-switch*** *from backpressured mode to backpressureless mode occur?* My policies use my first mechanism *local contention thresholds*. Section 4.3.4 answers the question: *How does AFC achieve correctness in the corner cases of interactions between routers in different modes?* My second mechanism *gossip induced mode-switch* ensures correctness in the corner cases. Section 4.3.5 focuses on the pipeline implementation of AFC, with a focus on AFC's third mechanism – *lazy VC allocation* proposed by Hong et al in [55]. Finally Section 4.3.6 discusses deadlock- and livelock-freedom for AFC networks.

4.3.1. Router Organization

I describe the AFC router organization by focusing on key similarities and differences of three router parameters with respect to the basic backpressureless and backpressured routers. First, the flits of the AFC router are wider because they have to include control information for both backpressured and backpressureless routers. Because AFC has to operate in the backpressureless mode at low loads, the AFC inherits hardware support for flit-by-flit routing from backpressureless routers (Section 4.2). This inheritance implies that links, buffers, and crossbars are wider to include sequence/packet numbers (for reassembly) and destination-node (for routing) in each flit. Similarly, each flit also carries a VC-identifier as required by backpressured routers. However, as I describe later in Section 4.3.5, lazy VC allocation reduces the number of bits that need to be propagated. AFC routers operating in the backpressureless mode continue to propagate the VC information along the next hop even though the router itself does not use the information (Section 4.3.5). Second, the AFC router inherits input buffers from backpressured routers. One may think that the inclusion of buffers results in AFC suffering energy/area penalty over backpressureless routers. However backpressureless routers incur significant performance and energy degradation at high loads. AFC's energy overhead is minimal at low loads because the buffers are bypassed when the AFC router is in the backpressureless mode. Further, AFC buffers use power gating [49] when in the backpressureless mode to avoid leakage energy. Note, such power-gating is practical in AFC because I power-gate at the granularity of an entire physical port's buffers. One may think that traditional backpressured routing can capture similar benefits by partially

power gating buffers at low loads (which is precisely when AFC is able to do power gating, as well). However, because VC buffers are implemented as circular buffers, different contiguous sets of buffer-entries may be active in different VCs. Therefore, per-flit gating will be needed, which is impractical. Further, such per-flit gating (or gating entire VC buffers) complicates neighboring credit management (or VC allocation). Moreover, reducing buffering in backpressured networks with multi-cycle links introduces credit-management pipeline bubbles. Finally, an AFC router is likely to be smaller than a full-blown backpressured router due to the smaller buffers afforded by *lazy VC allocation* (Section 4.3.5). Third, the AFC router includes credit propagation mechanisms to track per-VC buffer availability in neighboring nodes, as required by backpressured routing. Credit back-flows would unnecessarily add to the energy cost when the AFC router is in backpressureless mode where credits are meaningless. To avoid this energy overhead, I include a special control line (one bit) to indicate to adjacent nodes to stop/start credit tracking when the router switches to backpressureless/backpressured mode.

4.3.2. Forward mode-switch using local contention thresholds

Ideally, the forward mode-switch must occur when load levels are high enough that backpressureless routers are near saturation. The actual load at which this switch occurs may be dependent on global application traffic characteristics and hence independent of local injection rate. One option to detect near-saturation loads is to track the number of cumulative misroutes of flits and monitor if those exceed thresholds. However, that approach has two problems. First, it adds the overhead of modifying flits as they progress through each router. Second, high contention may be detected in an incorrect network region because a flit that passes through a high contention region may have undergone a number of misroutes, but the number may exceed the threshold only after it has already passed through the high-contention region and has reached a low-contention region. Therefore, AFC fundamentally requires local measures of contention.

AFC measures contention via local traffic intensity. AFC uses the number of network flits traversing through the router averaged over the previous 4 cycles, and further smoothed using an exponentially weighted moving average (EWMA) as a metric

of recent traffic intensity. Smoothing using EWMA was necessary to avoid frequent (and unnecessary) mode switches due to transient bursts of network activity. The measured traffic intensity is compared to an experimentally-determined local contention threshold, which is my first mechanism. The forward mode-switch is illustrated as the top transition in Figure 4.1. Because routers at edges and corners in a mesh have fewer ports, their thresholds are scaled accordingly. One may think that network traffic intensity could trigger false mode-switches because routers may observe high flit throughput without any link contention for "easy" traffic patterns (e.g., only near-neighbor communication). However, partly because real application traffic is not "easy" and partly because deflection induces further randomness, I found that the local contention thresholds were effective in detecting local load levels. Thus, my threshold is independent of application characteristics and is dependent only on the network con- figuration. Once triggered, the mode-switch is realized over *2L* cycles (where *L* is link latency). A mode-switch that begins in cycle *T*, will continue to receive any incoming flits in the input latches of the backpressureless mode. A notification to neighboring routers (via the credit-count start notification, as described in Section 3.3.1) lets them know that they should start counting credits in cycle $T + L$ even if the neighbors are in backpressureless mode. Because all flits received in the $(T + 2L - 1)^{th}$ cycle are guaranteed to have been dispatched in the $(T + 2L)^{th}$ cycle in the backpressureless mode, the backpressured mode can safely start from the $(T + 2L)^{th}$ cycle onwards (starting with the routing stage). Thus, any incoming flits that are received on or after the $(T + 2L)^{th}$ cycle are directed to the input buffers of the backpressured mode. Note that flits coming from previous backpressureless routers will still carry some VC information (Section 4.3.1) even though the routers do not allocate any VCs for the benefit of any backpressured routers downstream.

4.3.3. Reverse mode-switch

Just as the forward mode switch occurs under high-load conditions, AFC attempts the reverse mode-switch when the measured load falls below a different (lower) threshold. I use the two thresholds as a hysteresis mechanism to avoid frequent mode-switches in the case of a single threshold when the load hovers around the single

threshold. Instead, AFC maintains the previous mode when the load is between the high and low thresholds.

However, while performance/energy may indeed be maximized by switching to backpressureless mode as soon as load falls below a threshold, correctness requires that the reverse mode-switch cannot be initiated unless the buffers are empty. Without such a condition, flits remaining in buffers could be indefinitely trapped, leading to starvation. The reverse mode-switch is shown in the bottom transition in Figure 4.1. Once local buffers are empty, the router automatically starts operating in backpressureless mode in the very next cycle by bypassing incoming flits to the pipeline latches (for deflection) rather than to the input buffers. Subsequently, the switched router notifies its neighboring routers to stop accounting for credits. Upon receiving the notification, the neighbors simply set the buffer occupancy of the switched router to empty without waiting for actual credits. There is a time gap between when the switched router switches its mode and when the neighbors receive the notification (i.e., know that the switch has occurred). In this gap, the neighbors may have sent some flits to the already-switched router and decreased the router's credits not knowing that the switch has already occurred and that credit accounting has become unnecessary. However, because the discrepancy leads only to unnecessary accounting for a brief time period, no correctness issues arise.

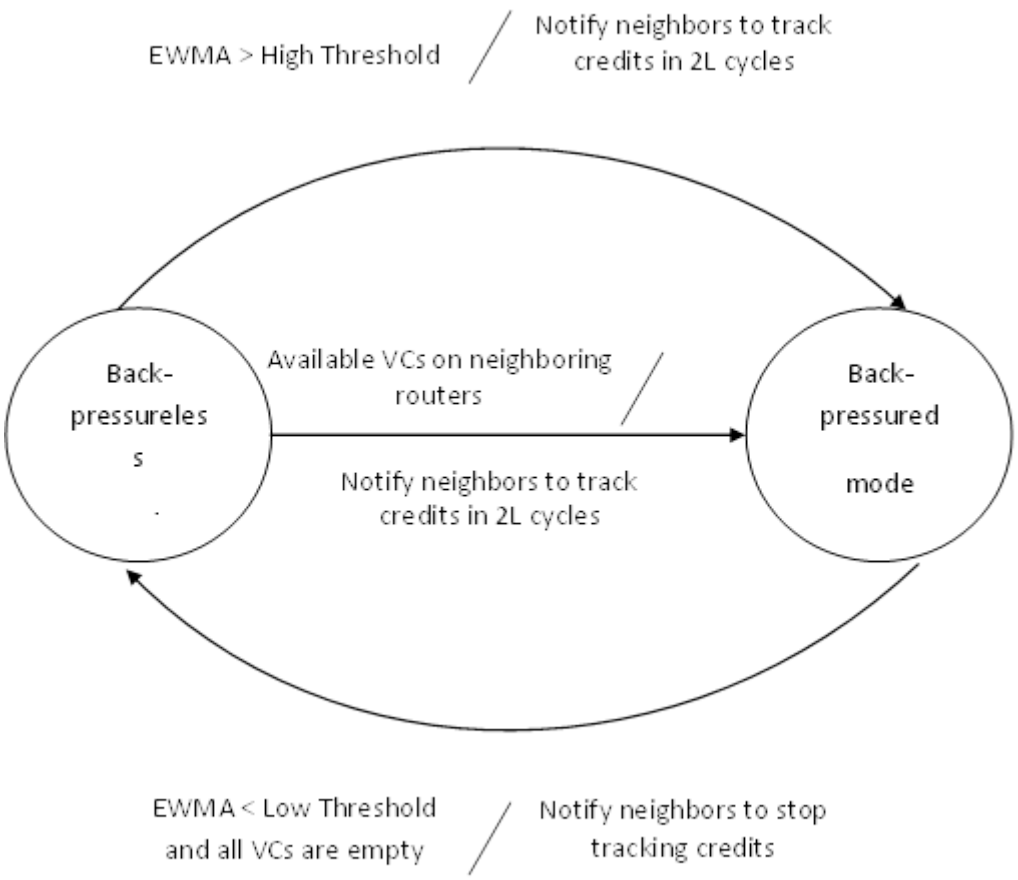

Fig 4.1AFC Mode transitions

4.3.4. Handling interaction among modes using gossip-induced mode-switch

Given that each AFC router switches modes independently, adjacent routers may be operating in different modes. Such a situation may occur as a stable state when there are large spatial disparities in load. Even when there are no steady-state spatial load variations, the situation may occur when there are transient hot spots.

Communication from a backpressured router to a backpressureless router is the easy case, because a backpressureless router is always willing to accept flits from neighboring routers. The difficult case arises when a backpressureless router communicates with a backpressured router because the backpressureless router may always dispatch flits on all

of its output links, but the backpressured router must be able to prevent incoming flits so that buffered flits are not overwritten.

AFC responds to this mismatch with a “scalpel and sledgehammer” approach wherein (1) AFC initially attempts to tolerate the mismatch with a lightweight response (the scalpel) which attempts to handle hot spots locally without spilling over to neighboring, low-contention regions; and (2) if the light-weight response fails, AFC responds with a heavyweight response (the sledgehammer) that guarantees correctness by expanding the backpressured region when the effects of contention cannot be contained. As part of the lightweight response, AFC ignores the mismatch as long as the downstream backpressured router has spare buffer capacity (as tracked locally using credits). Recall that the backpressureless routers begin to track credits as soon as their neighboring routers switch to backpressured mode (Section 4.3.2). If the credits indicate that a downstream router’s buffers are being exhausted (say, only X free buffers remain), then AFC employs the heavyweight response of a gossip-induced mode-switch, my third mechanism, wherein the neighboring backpressureless router is forced to switch to backpressured mode using the forward mode-switch (Section 4.3.2). The threshold X must be at least *2L* (I use *2L*) to allow sufficient time for a forward mode-switch (Section 4.3.2). The gossip-induced mode-switch is shown as the middle-transition in Figure 4.1.

4.3.5. Lazy VC allocation and its impact on the router pipeline

Recall from Section 4.2 that existing methods for achieving shallow pipeline depth in backpressured routers at low loads fundamentally rely on speculation. Such pipelines degrade to deeper router pipelines (3 stages, assuming look-ahead routing) at higher loads because of the need for VC allocation on a per-packet granularity to ensure that flits of packet stay together. However, with AFC’s use of flit-by- flit routing, VC allocation in the backpressured mode can be vastly simplified to operate at the flit-level and thus can be absorbed into other pipeline stages. Further, I leverage the simplification offered by lazy VC allocation [55] to reduce overall buffer requirements which compensates for a significant fraction of the energy overhead of AFC’s wider links.

To understand AFC's lazy VC allocation, examine the purpose of VCs and then examine the impact of flit-by- flit routing on VC allocation/deallocation. VCs serve two key purposes.

First, VCs are used to achieve deadlock-freedom by introducing VC traversal restrictions (rather than the more limiting physical channel traversal restrictions used by earlier router designs) that can prevent deadlock cycles. However, AFC employs dimension-ordered routing (DOR) which also provides deadlock-freedom, and hence AFC does not use VCs for this purpose. Nevertheless, AFC must still respect higher-level deadlock avoidance rules when allocating VCs. For example, if coherence requests and responses are in two different virtual networks, VC allocation must respect such restrictions. In such cases, one can view the overall virtual channel as a two-tuple consisting of a virtual network and the virtual channel within the virtual network.

Second, traditional VC flow-control prevents intermingling of flits of different packets which is necessary when a multi-flit packet is the smallest independently-routed unit. To prevent flit intermingling, VC allocation has to be globally coordinated to ensure that the following two rules are observed. (R1) No packet may be assigned a VC that has previously been assigned to another packet (but not yet freed). (R2) No two packets may be assigned the same VCs in the same cycle. When implemented as above, packets in different VCs are allowed to overtake/bypass one another, thus reducing HOL blocking.

Unlike traditional packet-switched, backpressured routers, AFC's backpressured mode uses flits (which may be viewed as single-flit packets) as the smallest independently-routed unit because packets may be broken up into flits at another backpressureless router. Because individual (independently routed) flits can be freely intermingled in input queues, VC allocation can be simplified by ignoring the two rules mentioned above. The first rule (R1) is impossible to violate because the busy state of a VC is used solely to prevent flit intermingling in multi-flit packets which is a non-issue for AFC. Further, rule (R2) can be ignored and any VC may be allocated to any flit (which may result in multiple flits being allocated the same VC in the same cycle). Such duplicate VC allocation is acceptable (from a correctness perspective) in AFC because the multiple flits will be serialized by the crossbar-switch anyway. Thus VC allocation

can be pre-computed locally (which may use simple round-robin or randomized allocation) thus completely eliminating VC allocation as a separate pipeline stage. However, duplicate VC allocation may cause unnecessary HOL blocking at the next (downstream) router because flits within a VC have to be routed in order. While the above optimization effectively eliminates VC allocation as a separate step, the possibility of duplicate VC allocation remains a performance problem.

AFC's lazy VC allocation resolves the above problem by assigning VCs at the downstream router by exploiting the observation that any VC allocation is legitimate. Lazy VC allocation views the K-flit input buffer SRAM structure as having K VCs per physical channel with a single flit buffer per VC. Further, unlike traditional credit backflow where credits at the upstream router are tracked on a per-VC level, AFC tracks credits on a per virtual-network level. As long as a virtual network is not full, there exists at least one VC with an unoccupied flit buffer. The upstream router dispatches flits to the downstream router without a VC allocation with only the virtual network identifier, which remains unchanged from one router to the next. Upon receipt, the flit is placed into one of the free flit-buffer entries (say, the i^{th} entry), thus lazily allocating the i^{th} VC to the flit. Note, free slots may be pre-discovered using simple daisy-chaining mechanisms and adds no latency to the critical path. Because all flits are placed in different VCs, the design avoids artificial HOL blocking (independent flits with the same VC allocation) altogether.

AFC's lazy VC allocation captures one additional advantage beyond the twin benefits of eliminating the VC allocation stage and minimizing HOL blocking. Because VC allocation is greatly simplified by flit-by-flit routing, AFC can increase the number of VCs without slowing down the VC allocation stage as would occur in traditional backpressured routers. To offset the energy increase of more VCs, AFC employs shallower buffers which suffice due to the following reason. While backpressured per-packet routing allocates an entire buffer for a packet so that some of the buffer slots are empty while the packet's flits are processed spread over time, flit-by-flit routing avoids this underutilization by allocating only one slot for a flit, thereby enabling shallower buffers. AFC reduces the total buffer size by a factor of 2 while matching the

performance of a tuned traditional backpressured router. AFC's shallower buffers mostly compensate for the energy overhead of its wider links. The activity in each of AFC's backpressured mode pipeline stages are summarized in the last row of Table 4.1. Because lazy VC allocation can easily fit (due to simple pre-computation) in the buffer-write stage, the AFC router can realistically operate in 2 cycles (as opposed to the generous 0-cycle VCA assumption for backpressured routers in Section 4.2).

4.3.6. Deadlock and Livelock Freedom

One may think that a combination of backpressureless routing (where DOR rules are ignored) and backpressured routing (where I rely on DOR for deadlock freedom) can lead to deadlocks. However, I can prove the deadlock freedom of AFC using the following two observations. First, deadlocks can occur in AFC only when all the flits in a deadlock "knot" are in routers that are backpressured. (If a single router is backpressureless, those flits are not blocked and hence can escape.) Second, given that all the routers are in backpressured mode, using DOR for the backpressured routers, I guarantee that backpressured routers are deadlock-free (i.e. escape paths always exist). (A similar property can be proved for non-DOR routers which use deadlock avoidance since escape paths will exist on some VCs.) The fact that those flits may have originally been deflected is immaterial.

Livelock-freedom is another important issue to address given AFC's use of backpressureless deflection routing at low loads, especially since AFC does not use (impractical) priorities to guarantee livelock freedom. Even in the absence of such priorities, deflection routing has been shown to be probabilistically livelock free [47]. Further, because AFC uses deflection routing only at low loads, the likelihood of a continuous chain of misroutes is even less likely. Thus, the probabilistic guarantees may be strengthened further. Recall from Section 4.2 that probabilistic guarantees are indeed strong guarantees.

4.4. Experimental Methodology

I evaluate AFC using Wind River's Simics 3.0 [20] full system simulation platform and the GEMS [21] timing models, which include an SMT-processor model (Opal), a

memory system model (Ruby) and an interconnection networks model (Garnet) [54]. **Simulated Machine Configuration**: Table 4.2 summarizes the key parameters of my simulated machine. Simulating a 16-core system (with multi-threading, as exemplified by many recent "CMP of SMTs" [56], [57], [58]) proved infeasible because of long simulation times as simulator state spilled out of memory to swap-space. I employ conservative scaling to simulate a 3x3 network with 9 nodes. The scaling makes my results conservative because the saturation throughput for backpressureless routers is higher when the network is smaller. Based on the arguments in Section 4.2 and Section 4.3.5, all routers are simulated with 2-cycle pipelines (see Table 4.1). My configuration (number of VCs and buffer-depths) is energy-optimized for the backpressured base case. Adding more VCs (or increasing buffer-depths) resulted in no significant performance improvement. I used 32-data bit flits in each direction as I found that they were energy-delay-squared optimal. The control link widths were chosen so that VCs, destination nodes, flit-numbers, and global MSHR identifier could be encoded. Such encoding required 9 bits for backpressured networks, 13 for backpressureless networks, 17 for AFC. Thus, the total flit width, including data and control lines, were 41 (backpressured), 45 (backpressureless) and 49 (AFC) bits. These flit widths are reasonable because (a) they correspond to fairly wide 80–94 bit buses for full-duplex communication, (b) they are similar to the on-chip network in Intel's Teraflops research chip [59], (c) wider widths will reduce AFC's overheads, and (d) wider widths will cause superlinear growth in crossbar area.

AFC Parameters: AFC uses 8 VCs for each of the two virtual control networks and 16 VCs for the virtual data network with 1-flit deep buffers in its backpressured mode. The per-physical-channel buffer size is therefore 32 (= 8 × 2+16) flits in comparison with the baseline packet switched network which uses a 64-flit buffer (=4×8+2×2×8). Recall, the reduction in buffer-size is enabled by lazy VC allocation. The local contention thresholds for the forward (reverse) mode switch are set to 1.8 (1.2) for the corner routers, 2.1 (1.3) for the edge routers and 2.2 (1.7) for the other routers. The weighting factor for EWMA is 0.99 (i.e., the moving average update computation is $mnew = 0.99 * mold + 0.01 * l$ where l is the average load over the past four cycles).

Energy Modeling: The Garnet network timing model is integrated with callbacks to the Orion [60] network energy model to report energy dissipation. I model all the key additional hardware for backpressureless routers and for AFC including flit-latches and additional control links. Because the receive-side buffering for flit reassembly is required for both backpressured and backpressureless routers [45], and because they are associated with MSHRs I exclude the receive-side buffers from network energy. The MSHR buffer sizes do not vary with flow-control mechanism since they are provisioned for the worst case (e.g., in the worst case, all-but-one flit corresponding to each outstanding MSHR entry arrives at the node). I used the parameters for 70nm technology with Vdd of 1.0V and 3GHz frequency. I assume 2.5mm links. I realistically assumed 90% effective power gating when AFC power-gates buffers in its backpressureless mode. Finally there are previously proposed optimizations that target crossbar dynamic energy. For example, an aggressive variant of express virtual channels [44] proposes to use wires that bypass the crossbar switch for packets that traverse express virtual channels that proceed along the same dimension. Note, such an orthogonal optimization can be grafted on to any of backpressureless, backpressured or AFC router by adding bypass paths between appropriate pairs of ports and letting flits that traverse the corresponding ports to use the bypass paths instead of the crossbar. As such, I do not consider this optimization in my comparisons.

Table 4.2Workloads: Description and characteristics

Commercial Workloads
Apache: version 2.2.0, a static web server workload with repository of 20,000 files (~500 MB). SURGE is used to generate web requests by stimulating 4500 clients with 25ms think time between requests. Inj Rate = 0.78
Online Transaction Processing (OLTP): models database transactions of a wholesale parts supplier. I use PostgreSQL 8.3.7 database system and DBT-2 test suite which implements TPC-C benchmark. I reduced number of items and districts per warehouse and customers per district to allow a larger number of warehouse. I use a database of 25,000 warehouses (˜5GB). I simulate 300 concurrent database connections. Inj. Rate = 0.68
SPECjbb: version 2005, Java-based 3-tier client/server system workload with emphasis on the middle tier. Java server VB version 1.5 with parallel garbage collection. I simulate a system with 90 warehouses. Inj. Rate = 0.77
SPLASH-2 Workloads
Barnes: implements the Barnes-Hut method to simulate an N-body problem. I use 8 threads with a problem size of 512 particles. Inj. Rate = 0.1
Ocean: simulates ocean movements based on eddy and boundary currents with contiguous partitions to enhance data locality. I use 8 threads with a problem size of 34x34 grid. Inj. Rate = 0.19
Water-nsquared (Water): simulates water molecules by solving Newtonian equations using a predictor-corrector method in each time step. I use 8 threads with a problem size of 64 molecules for one time step. Inj. Rate = 0.09

Workloads: While open-loop simulations have some value, relying solely on them is problematic because they set injection rates to arbitrary values which may or may not correspond to real workloads. Trace-driven evaluations do not include the feedback effect

of the network on execution time. To avoid these problems, the majority of my experiments use execution times on multi-threaded applications to evaluate AFC. I do not use multi-programmed, sequential workloads because they lack coherence interactions which fundamentally change the network traffic. My benchmarks include three high-load/commercial and three low-load/scientific multi-threaded applications (see Table 4.3). Table 4.3 shows the injection rate achieved by each benchmark (in flits/node/cycle). I run the commercial benchmarks for a fixed number of transactions after adequate cache warmup (see Table 4.4). I scale scientific workloads from SPLASH-II benchmark suite [61] to run to completion. I also include an experiment with synthetic random traffic to highlight key performance/energy characteristics of AFC. I repeat all simulations multiple times to account for statistical variations.

Table 4.3 Simulation parameters for commercial workloads

Workload	System Warmup (transactions)	Cache Warmup (transactions)	Measurement (transactions)
Apache	2 million	20,000	600
OLTP	0.1 million	5,000	50
SPECjbb	1 million	50,000	3,000

4.5. Results

Recall that, for the low-load applications, backpressureless networks are expected to consume less power than the backpressured networks. In contrast, for high load applications backpressured networks are expected not only to consume less energy than backpressureless networks but also to outperform backpressureless

networks.

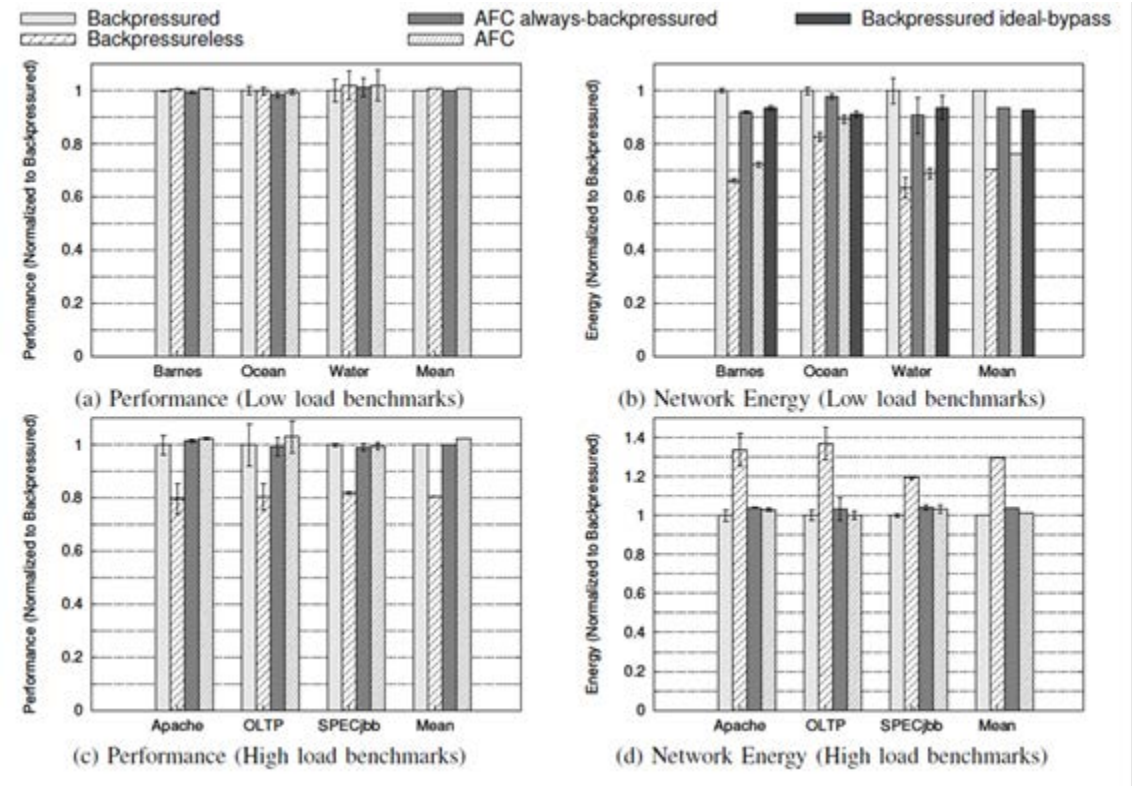

Fig 4.2 Performance and energy robustness

4.5.1. Performance and Energy

Recall, AFC's key goal is to match the performance and energy of the better flow control mechanism at both low and high network loads. The four figures in Figure 4.2 plot the normalized performance (left) and network energy (right) at low loads (top) and high loads (bottom). For the performance graphs, higher is better and for the energy graphs, lower is better. Each graph includes the set of benchmarks (groups of bars) on the X-axis with the appropriate metric (performance or energy) on the Y-axis. The Y-axis numbers are normalized to that of the baseline backpressured network. Three of the bars within each group correspond to the three flow-control mechanisms being compared. I also show one other comparison. AFC combines mechanisms for adaptively switching between flow-control modes as well as mechanisms for optimizing the flit-by-flit

backpressured mode with lazy VC allocation. To isolate the effects of the two sets of mechanisms, I include an AFC router which is always in the backpressured mode (called "AFC always backpressured"). Finally, rather than compare AFC against the each of the many proposed buffer energy optimizations, I show a packet-based router in which all buffer dynamic energy is eliminated (called "Backpressured ideal-bypass"). This serves as a lower bound on energy for techniques that elide buffer reads (but not writes) [43] as well as those that elide a fraction of both buffer reads and writes (Express virtual channels [44]). I show this bound only for the low load, energy graph, which is where it is relevant. Each graph also includes the geometric mean (rightmost bars). The variance bars indicate the standard deviation of multiple runs of the benchmark.

I make four key observations. First, at low loads, flow control has no meaningful impact on performance (see Figure 4.2(a)). This is not surprising given my expectation that lack of contention implies that there is little misrouting in backpressureless routing. AFC, which operates in backpressureless mode at low loads achieves similar performance. The backpressured router and AFC's always-backpressured router are also similar in performance.

Second, flow control does have a big impact on energy (see Figure 4.2(b)). Backpressureless, which eliminates buffers and thus, all buffer energy, consumes the least energy. AFC, which is largely in backpressureless mode, achieves within 9% of backpressureless. This gap mostly comes from my assumption that power-gating the buffers eliminates only 90% of their static power. The basic backpressured router, without any buffer energy optimizations, is the most energy consuming (42% more than backpressureless). More interestingly, even backpressured-ideal-bypass, where all dynamic buffer energy is elided, is significantly worse (32%) than backpressureless. This result strengthens the argument that dynamic buffer power optimizations have fundamental limitations at low loads, where static power dominates. Further, the skew in favor of static power will only worsen as I move to future technology generations.

Third, at high loads, backpressureless routing suffers a significant degradation in performance relative to backpressured routing (19% on average, see Figure 4.2(c)). This degradation is due to excessive misrouting. AFC, which is largely in the backpressured

mode, achieves comparable performance (within 2%). Not surprisingly, AFC-always backpressured is also very similar. Note that the backpressured router, which assumes a fixed 2-cycle pipeline with 0-cycle VCA subsumes previous pipeline optimizations for backpressured routers (Section 4.2).

Fourth, on the energy front, the behavior in terms of performance is mirrored in energy. Backpressureless, which has the worst performance, also dissipates the most energy, being 35% higher than backpressured, which is the least energy configuration. In contrast, AFC incurs a modest energy overhead (2% on average, 3% worst-case) compared to backpressured.

In summary, AFC matches the performance of better of both backpressureless and backpressured flow control at both high and low loads. It approaches the better of the two in terms of energy as well (within 3% at high loads and within 9% at low loads). In contrast, non-adaptive backpressureless flow control incurs a 19% performance penalty and a 35% energy penalty at high loads. Conversely, at low loads, non-adaptive backpressured flow control incurs an energy penalty of 32% (on average) even with ideal buffer bypassing.

Energy breakdown: Figure 4.3(a) and Figure 4.3(b) plot the normalized energy (normalized to backpressured router's energy, Y-axis) for my low load and high load benchmarks (X-axis), respectively. Each flow-control mechanism (bars within group) is shown for each benchmark. Further, the overall network energy is shown partitioned into buffer energy, link energy and other router energy (which includes crossbar energy and arbiter energy).

For low-load applications (Figure 4.3(a)), all three benchmarks exhibit largely similar energy profiles. The breakdown for backpressured routers indicates that buffer energy is significant, even in the case with the smallest proportion of buffer energy (ocean). In contrast, backpressureless routers eliminate all buffer energy for a modest increase in link energy. Because AFC largely stays in backpressureless mode, it too eliminates most buffer energy. Finally, though AFC always-backpressured reduces some buffer energy because it uses half as much buffer space as the backpressured router, buffer energy remains a significant fraction.

On the other hand, at high loads, backpressured mode achieves the lowest energy across all benchmarks. Backpressureless routers incur a significant link energy penalty due to excessive misrouting as mentioned before. There is little difference between AFC and AFC-always-backpressured because AFC largely stays in the backpressured mode. I observe that the overall energy penalty of AFC (relative to backpressured) is the difference between increased link energy (due to wider flits) and reduced buffer energy (due to lazy VC allocation).

Mode duty cycle and spatial variation: I measured the fraction of time spent by AFC in the two modes for all my workloads. Four of the six benchmarks were uniformly high or low load without any variation in time. For example, water and barnes were in backpressureless mode 99% of the time. Similarly, specjbb and apache were in the backpressured mode more than 99% of the time. The other two benchmarks exhibited a small amount of variation. For example, routers spent 7% of ocean's execution time in backpressured mode and 5% of oltp's execution time in backpressureless mode.

Interestingly, although my runs did not see any gossip induced mode-switches, I did see them in an open-loop network experiment which created hotspots. Recall, gossip induced mode switch is required for correctness and is therefore justified even if my runs do not exercise it. The lack of gossip-induced mode switching in my runs indicates that either transient hot spots did not develop because the network load was uniform, or if any transient hot spots developed, they were tolerated by the "scalpel" (Section 4.3.4) wherein adjacent routers did not see any backpressure that would have forced them to switch modes (i.e., the hot-spot would spread).

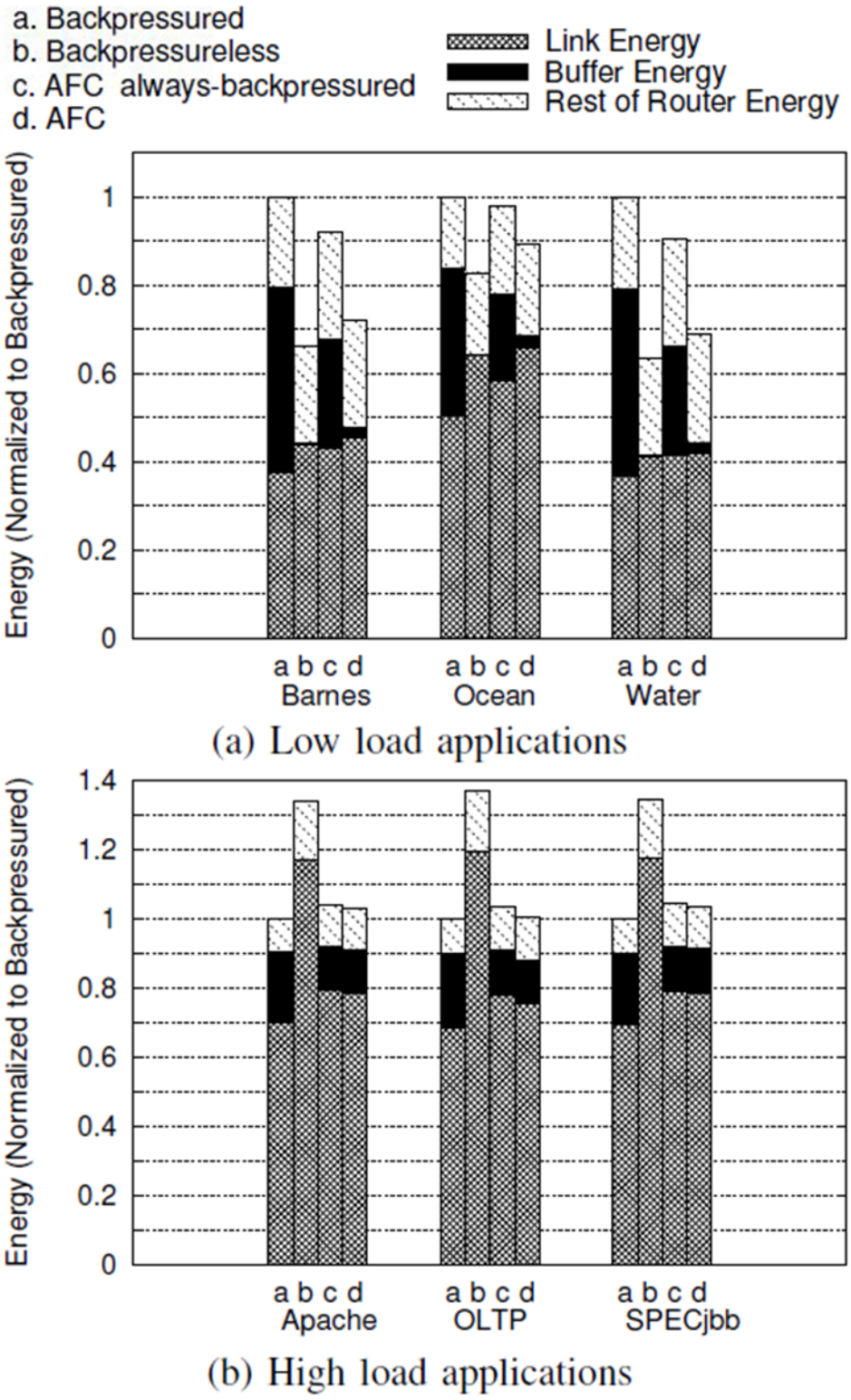

Fig 4.3 Network energy breakdown

4.5.2. Open-loop evaluation for spatial variation

Because of the near-perfect spatial uniformity of load in my runs, I used synthetic traffic with open-loop simulation to simulate spatial load variation. The configuration is designed to mimic a consolidation workload in an 8x8 multicore in which a different application runs in each quadrant. One quadrant of the network injected packets at a fixed high rate (0.9 flits/node/cycle) and the other three quadrants injected packets at fixed low rates (0.1 flits/node/cycle). The destinations were chosen such that traffic injected in a quadrant stayed within the quadrant (except possibly due to misrouting).

In the absence of variation, AFC can only approach the best of either backpressured or backpressureless routers. However, with the spatial variation described above, AFC was the best energy configuration because neither backpressured (9% more energy than AFC) nor backpressureless (30% more energy than AFC) flow control was robust in handling the load variation. I also observed that (1) backpressured and AFC achieved 33% lower latencies than backpressureless in the high-load quadrant, and (2) the high load quadrant had an adverse impact on a neighboring low load quadrant's latencies because of misrouting.

Other results: Experiments with open loop, uniform random traffic injected at various rates revealed that (1) all flow-control techniques achieve similar latencies at low loads (2) AFC and backpressured networks achieve near identical saturation throughput (whereas backpressureless saturates at lower offered loads).

4.6. Related Work

There is a rich body of work on each of the two flow control mechanisms. Deflection routing, first proposed in [48], has seen several variants implemented in real machines and in research prototypes [62], [63], [47], [64]. Deflection routing has also been studied extensively [65], [66]. More recently, researchers have refocused on deflection routing as an attractive option for energy-constrained on-chip networks [67], [68], [69], [45]. All the above variants of deflection-based routing either drop packets or suffer from high latencies at saturation (which is typically at lower loads than with backpressured routers). In contrast, AFC adaptively changes the flow-control to enable backpressured mode of operation.

Similarly, there has been extensive research on backpressured networks with credit-based flow control. Since my focus is on optimizing energy and latency in backpressured networks at low loads, I discuss only the work relevant to those goals. For example, Wang et al. propose a technique to bypass buffer-reads under low loads when there is only one flit in the buffer [43]. While such techniques do help reduce dynamic energy, they are not as energy efficient as eliminating all buffer dynamic energy and most static energy (using power-gating) as in the backpressureless mode of AFC, even at low loads. There have been techniques proposed to target leakage power of buffers by placing buffers in inactive modes [70]. In general, they require fine-grained (flit-by-flit) power-gating which may be unviable, especially given my small buffers. Techniques that speculatively overlap key pipeline stages (e.g., VC allocation and switch arbitration in [51]) attempt to avoid the latency penalty of a backpressured router's pipeline stages at low loads. Hong et al's lazy allocation goes one step further and removes the dependence between VC allocation and switch allocation.

While single-cycle routers have been proposed [53], [46], they will likely need a slow clock to accommodate both switch arbitration and switch traversal. For example, the router in [53] employs speculative switch arbitration in parallel with switch traversal. However, the router design assumes that misspeculations can be caught and recovered from in the same cycle. Effectively, this assumption implies that the router can ensure a conflict-free switch arbitration and switch traversal in the same cycle. In the case of SCARAB [46], switch arbitration and switch traversal are non-speculative and must fit in a clock cycle.

4.7. Conclusion

As the microprocessor industry packs more cores into a chip, multi-hop interconnection networks are likely to be used as the on-chip communication fabric. Network performance has a direct impact on overall system performance. In turn, flow-control mechanisms have a first order impact on the performance and energy of networks. Two widely-studied flow-control mechanisms – credit-based backpressured flow control and backpressureless deflection flow control – have their own particular network load "sweet spots" where they operate well. Unfortunately, they incur significant

performance/energy penalties at loads outside their sweet spots. For example, backpressureless networks achieve low energy at low network loads, but suffer from excessive misrouting at high loads, which leads to poor performance and energy. Similarly, backpressured networks are energy-efficient and achieve high throughputs at high network loads that backpressureless networks cannot reach. However, backpressured routers incur a energy penalty at low loads. If network loads for real applications were predominantly in either high- or low-load region, one of the flow control mechanisms would suffice. Unfortunately, workload characteristics are not limited to the “sweet spot” region of any single flow control mechanism.

I along with my colleagues Yu-Ju Hong, Mithuna Thottethodi and T. N. Vijaykumar propose Adaptive Flow Control (AFC) – a robust flow control mechanism with a wide sweet spot that spans high and low loads. AFC routers operate in backpressureless mode at low loads and as backpressured routers at high loads. Consequently, AFC avoids the significant energy/performance penalties that each of the two flow control policies incur when operating outside their sweet spots. Evaluation with a suite of multi-threaded commercial and scientific/engineering workloads reveals that AFC’s performance and energy are close to those of the better of backpressured and backpressureless routers. As the number of cores continues to scale, and as the mix of applications grows more diverse, AFC’s performance and energy robustness will be increasingly important.

5. APSLIP

5.1. Introduction

The queuing discipline employed in the on-chip network router has a first order impact on both latency and throughput of the network. Routers can queue flits either at the input ports or the output ports. However, input-queued routers suffer from head-of-line (HOL) blocking which significantly degrades performance [71]. In contrast, output-queued routers are free of HOL blocking but naïve implementations require write bandwidth to the output queues to scale with the number of input ports for the cases where flits from multiple input ports are destined to a single output port. This “speed up” of the output queues is hard even for a few input ports [50]. To address this issue, Karol et al. in [72] propose the virtual output queuing (VOQ) architecture for routers. VOQ creates as many queues at each input port as there are output ports. Because each queue corresponds to a single output port, VOQ completely eliminates head-of-line blocking without the need for speedup of the switching fabric. However, on-chip networks require flow control which raises some issues for VOQ which I address using known techniques.

To be effective, however, the VOQ scheme requires a sophisticated switch allocation algorithm which can support high network throughput. A low throughput switch would throttle the network and render the VOQ scheme useless. McKeown proposes the iSLIP switch allocation algorithm in [73] which approaches close to a 100% network throughput. VOQ routers along with the iSLIP switch allocation algorithm have been used extensively in Internet routers. Internet routers can exploit VOQ/iSLIP because the lack of flow control does not matter; they can drop packets upon congestion. In contrast, on-chip network routers cannot gain from the iSLIP algorithm which necessitates a slow clock. Clock speeds are more critical than Internet router clock speeds where router delay is a small fraction of the long end-to-end delay (e.g., 40 ms). Pipelining iSLIP to achieve

fast clock is challenging due to dependencies which is the main problem I address in this chapter.

An alternative to pipelining is to adapt per-packet switch allocation which reduces the importance of fast allocation by decreasing the frequency of allocation from per-flit to per-packet. In per-packet allocation, a packet holds the allocated switch port until all the packet's flits are transmitted. Such allocation enables the use of sophisticated (and slow) switch allocators, as employed in Internet routers, where slow clocks are acceptable. However, there are two key disadvantages for on-chip networks. First, per-packet allocation requires either full-packet buffering (which can add significant area/power overheads) or reservation of unused links when packets are spread over multiple routers (which can exacerbate tree-saturation and hence hurt performance). Second, because on-chip networks have a large number of small, single-flit control packets, per-packet switch allocation is no better than per-flit switch allocation. Packet chaining [83] ameliorates this problem by chaining multiple small packets together whenever possible; but at the cost of additional hardware complexity to detect chaining opportunity and duplicate allocators to exploit the opportunity.

Due to the above problems with VOQ and iSLIP, current on-chip network routers employ input queuing implemented via virtual channels (VCs) to alleviate HOL blocking along with simple switch allocation algorithms which are pipelined for throughput. However, the simple algorithms (e.g., SPAA [74]) offer no theoretical guarantees that they can achieve full (100%) network throughput, unlike iSLIP.

I propose apSLIP which combines VOQ and adaptive-effort, pipelined iSLIP to achieve higher network throughput than the current combination of input queuing and simple switch allocation algorithms. While apSLIP can work with per-flit or per-packet allocation, I focus on per-flit allocation due to its lower hardware overhead.

To provide flow control with VOQ, I observe that in traditional networks, the source router allocates the VC at the destination router and tracks the VC's occupancy for flow control. In VOQ, however, the destination virtual output queue is determined at the destination router, unknown to the source router. To address this problem, I utilize look-ahead routing [50] where the destination's output port and therefore the virtual output

queue are known at the source router. Alternatives to flow control, such as dropping or deflecting flits, perform worse at high network loads [46], [45]. In addition to flow control, VCs can also provide deadlock freedom for which I use the well-known alternative of dimension-ordered routing (DOR).

To address the main problem of pipelining iSLIP, I propose three novel ideas. Pipelining iSLIP is challenging due to two dependencies amongst its three phases (natural pipeline stages), which cause RAW hazards. The first hazard involves resending requests for flits before the outcome (grant/no-grant) of the previous request for the same flits is known. Such re-sent requests would be superfluous if the earlier request is granted and the corresponding flit dispatched. Such superfluous requests may then receive output grants which constitute lost opportunity for other contending flits. My first idea is based on the key observation that with VOQ and at high network loads, each virtual output queue will have more than one flit in the common case. Therefore, there will almost always be other flits waiting in the same queue to avail a grant for a superfluous request. I emphasize this VOQ-iSLIP synergy that the grant can be availed easily only in VOQ where all the flits in the queue are destined for the granted output which is not the case in input queuing where finding a flit in an input queue for the granted output is hard. Therefore, combining iSLIP with input queuing instead of VOQ would not achieve the same effect.

The second hazard is a RAW hazard that arises because priority-counters used for round-robin arbitration are written in stage 3 but read in stage 2. Because the priority counters hold metadata and not program data, I ignore the RAW hazard and use stale metadata without causing correctness problems. However, such a strategy does cause performance degradation because of double-booking of resources. I overcome this double booking by separating the arbitrations into odd and even streams which amounts to privatizing the priority counters (a separate set of counters for each stream instead of one-set of counters for all arbitrations).

Pipelining iSLIP fundamentally enables another optimization in the switch allocator by exploiting a key feature of iSLIP. iSLIP is one of the maximal-matching allocators that can achieve higher-quality matching at higher effort via more iterations of the

matching algorithms. Unpipelined, multi-iterative iSLIP implementations are worse than single-iteration implementations when it comes to clock speed. However, my pipelining can achieve a 2-iteration, 6-stage pipelined implementation at a fast clock. While the second iSLIP iteration is useful at high network loads (where the increased bandwidth helps reduce queuing latency), the extra latency hurts performance at low loads (where there is no increase in throughput). To address this issue, I propose my third idea of an adaptive-effort allocator that adapts the pipeline depth between one and two iterations depending on the injection rate to achieve low latency at low loads and high bandwidth at high loads.

In summary, the chapter's contributions are:

- I pipeline iSLIP by addressing two key hazards:
    - For superfluous requests, I leverage the VOQ architecture which naturally enables easy availing of the corresponding grants
    - For priority-counter hazard, I use stale priority values and avoid the resulting double booking by privatizing the priority counters and separating the arbitration into odd and even streams.
- I propose *apSLIP*, an *adaptive-effort pipelined iSLIP* which adapts between low-effort, low-latency matching at low loads (i.e., one iteration in three stages) and high-effort, high-bandwidth matching at high loads (i.e., two iterations in six stages).

Comparisons with several switch allocators using full-system and trace-driven simulators running commercial and scientific workloads show that apSLIP with per-flit allocation outperforms (a) an aggressive 2-cycle per-flit allocator by 20% on average for high-load benchmarks without affecting the low-load benchmarks and (b) idealized packet-chaining (with per-packet allocation) by 9% on average for high-load benchmarks while using smaller buffers and avoiding duplicate allocators.

The rest of the chapter is organized as follows. Section 5.2 discusses related work. Section 5.3 provides a brief background on router queuing disciplines and iSLIP. Section 5.4 describes apSLIP's details. Section 5.5 describes my experimental methodology and Section 5.6 presents experimental results. Finally, Section 5.7 concludes the chapter.

5.2. Related Work

Alternatives to pipelining iSLIP are: (1) bypass the router, (2) reduce router latency to 1 cycle, (3) make switch allocation unimportant, and (4) improve switch allocation algorithm. Proposals for the first option speculatively exploit the lack of resource contention at low and near-zero loads [44] [84] to allow flits to bypass most of the router and incur only wire-delays. The SMART router extends this further to achieve multi-router traversal with only wire-delays by leveraging multi-drop transmission lines [85]. In general, such speculative techniques degenerate to full router latency at modest and high loads. The question then arises as to what loads are reasonable. I observe that network load (in flits/node/cycle) can be made arbitrarily low by making network links arbitrarily wide. However, practical link/bus-widths real products have remained in the 16-64-bit range [86] [87] because of two reasons: (1) The actual widths for point-to-point links are significantly higher than the nominal flit width because of differential signaling (2x increase) and the need for two unidirectional links for bidirectional communication (another 2x factor). (2) The compute overhead of CRC, needed for reliable transmission, increases linearly with the link width limiting the clock speed and hence bandwidth. For example, QPI, Intel's on-chip network for current and future products, uses 84 signals to communicate the 20-bit phits (16-data bits + 4 CRC bits) [86]. Shared buses, which do not have to offer duplicate unidirectional links, still have significant bloat. For example, the 64-bit front-side bus (FSB) uses 150 pins. Consequently, though metal layers provide ample connectivity, the above reasons limit the effective width of practical links. I assume 32-bit flit widths which uses approximately 150-bit wide links, in line with current and future real products. With such reasonable link widths, I find that the network loads with commercial workloads are high enough that such speculative techniques do not work well in practice. As such, apSLIP significantly outperforms the techniques (Section 5.6.1).

The second option includes many shallow-pipelined or even single-cycle router proposals [46] [53]. There are two ways in which the entire router can fit within a single cycle. First, the critical path through the router is truly reduced by eliminating key dependencies and enhancing circuit-level parallelism. In general, modern router designs

do not have superfluous dependencies that may be non-speculatively eliminated. Alternately, the second possibility is that even though the critical path is unchanged, the clock happens to be slow enough to accommodate the entire critical path. Such a design gains a marginal latency advantage over a pipelined alternative because of latch over heads in the pipelined design. The latency advantage comes at the cost of reduced bandwidth and is limited only to low loads. At high loads the low bandwidth significantly degrades performance compared to a pipelined alternative with a faster clock. Additionally at low loads, there is not much communication and hence little opportunity to impact overall performance so that the latency advantage does not matter much. At high loads, however, queuing delays dominate router delays, which implies the pipelined design will achieve both better latency and better bandwidth. Not surprisingly, my comparison with an ideal, single-cycle router shows that apSLIP significantly outperforms the router (Section 5.6.1).

As discussed in Section 5.1 per-packet switch allocation (e.g., packet chaining [83]) reduces the importance of fast allocation – the third option – but requires full packet buffering to avoid severe performance degradation. This requirement can lead to large buffers and area/power overheads. For example, assuming 7 ports (4 network ports + 3 local ports), a coherence protocol that uses 3-5 virtual networks, 32-bit flits, 18-flit packets (assuming 64-byte cache blocks), and 8 VCs per virtual network, a per-packet design requires between 12-20 KB buffers per router. In contrast, per-flit routing may use fewer flit buffers (say 4-8 flits/VC) thus reducing buffer requirements by 2X-4X (4.4-8.8 KB per router).

For the fourth option, TS-router [88] proactively avoids scheduling conflicts by using knowledge of future (conflicting) flits. Input ports where flits are expected in the future are prioritized for switch allocation to evacuate older flits before the scheduling conflict occurs (on the arrival of the future flit). This *anticipatory evacuation* policy is effective only at low loads when input queue occupancy is low and thus evacuation is feasible. At medium/high loads, when there are higher numbers of flits, it is impossible to evacuate all flits in time to avoid scheduling conflicts. Consequently, apSLIP significantly outperforms even a 1-cycle TS-router (Section 5.6.1).

5.3. Background

In this section, I discuss queuing discipline in routers and iSLIP and its variants.

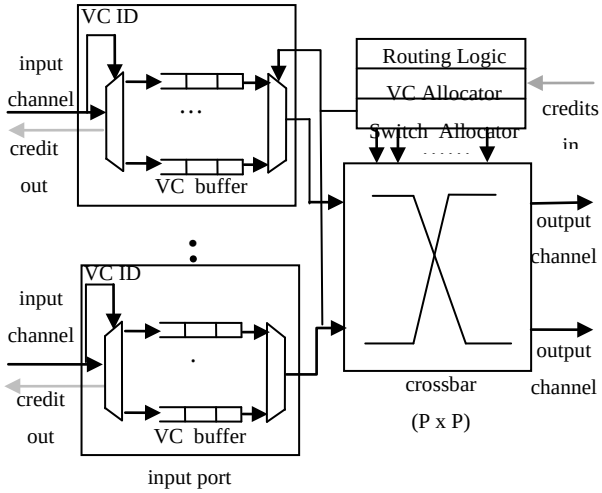

Fig 5.1Architecture of router using virtual channels

5.3.1. Input Queuing

Karol et al. [71] showed that the throughput of an NxN port input-queued switch with FIFO queues, under certain conditions, will be limited to just (2-√2) = 58.6%. The underlying cause of this limitation is HOL blocking, where flits are delayed by other flits ahead in line destined for a different output port. HOL blocking occurs only in the case of FIFO queues and many techniques have been proposed for reducing HOL blocking by using non-FIFO queues. One of the most prevalent techniques for reducing HOL blocking is virtual channel flow control proposed by Dally et al. [75]. As shown in Fig 5.1, a virtual channel (VC) is associated with a buffer which can hold flits of a single packet and other state information. Multiple VCs share the bandwidth of a single physical channel. Hence VCs act like multiple FIFO queues at each input of the router. If flits of one packet (hence one VC) are blocked, the input port can transfer flits from another packet (another VC) hence mitigating HOL blocking. When the packet is fully transferred, the router can allocate the VC to another incoming packet. While VCs can ameliorate HOL blocking, they cannot completely eliminate HOL blocking. Each VC can

hold the flits of just one packet and there are a fixed number of VCs at each input. Therefore, a scenario where there are flits of more packets vying for the same input port of a destination router than there are VCs on that input port gives rise to HOL blocking. At the source router, flits that would have gone to a particular output on the destination router are delayed because all the VCs on the input of the destination router are allocated to flits destined for another output which is essentially HOL blocking.

Mukherjee et al. [74] perform a comparison of various switch allocation algorithms for VC based flow control. They propose the Simple Pipelined Arbitration Algorithm (SPAA) and showed its superiority to unpipelined iSLIP and unpipelined Wave Front Algorithm (WFA) [76]. While both iSLIP and WFA can reach higher throughput than SPAA, they are not pipelined and cannot compare in performance with pipelined SPAA at a fast clock. However, SPAA sacrifices powerful matching of input to output ports in favor of pipelineability

5.3.2. iSLIP Operations and Pipeline Hazards

Proposed by McKeown in [73], iSLIP is an allocation algorithm that provides lower latency as compared to PIM in general and can theoretically reach a 100% network throughput. I enumerate the key steps of iSLIP below:

1. Request (RQ) stage: Each input port sends requests to every output port for which it has a flit.
2. Output Arbitration (OA) stage: Each output port selects a request based on a private counter and informs the corresponding input port. Note the counter is not incremented at this stage.
3. Input Arbitration/Counter Update (IA/CU) stage: In an input port receives grants from multiple output ports, it selects on based on a private round robin counter. The input and output port both increment their counters.

Fig 5.2 illustrates the unpipelined operation of the iSLIP allocator for two flits. There are two cases of dependencies. First, the RQ stage for subsequent allocation attempts uses information on successful matches from the previous allocation to ensure that successfully matched flits do not continue to assert requests (solid arrow in Fig 5.2). Second, the priority counters used for round-robin arbitration are written in stage three

and read in the OA stage of subsequent allocations (dashed arrow in Fig 5.2). Pipelining iSLIP reveals that each of these two dependencies translate to two RAW (read-after-write) hazards (Fig 5.3).

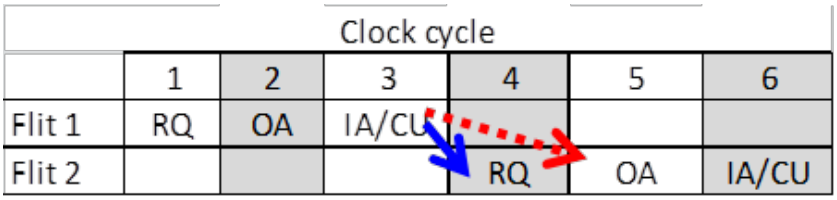

Fig 5.2 Value communication in unpipelined iSLIP

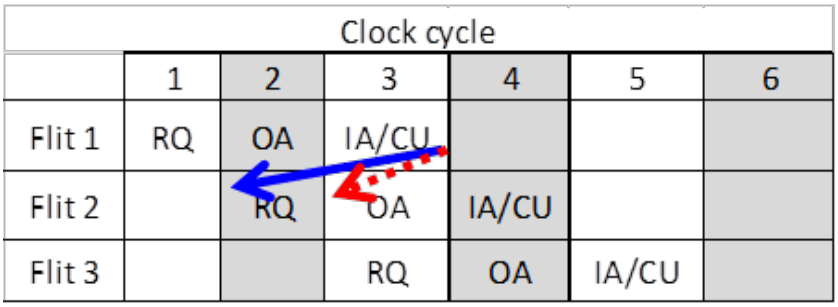

Fig 5.3Hazards exposed by pipelining iSLIP

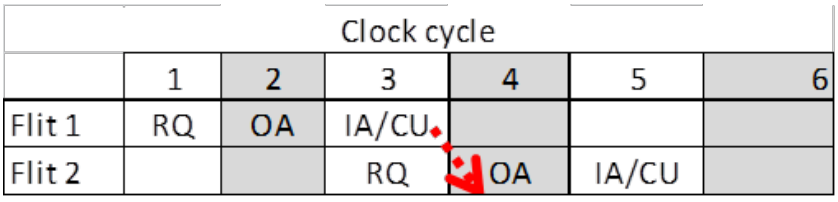

Fig 5.4 Inter-iteration pipelining in Tiny Tera

5.3.3. VOQ and its variants

In contrast to VCs which map input FIFO queues to packets, VOQs map FIFO queues to the output ports of the router thus completely eliminating HOL blocking (see Fig 5.5).

As I mention in Section 5.1, while implementing virtual output queuing is non-trivial in a flow-controlled network, VOQs have been widely adopted in Internet routers where flow control is not required. Researchers have proposed several variants of the powerful multi-iterative iSLIP algorithm to provide high-throughput switch allocation in virtual output queued internet routers. Nick McKeown proposes pipelining across different iterations of iSLIP in the Tiny Tera Internet router to reduce the latency of a single round of multi-iterative iSLIP allocation. The Tiny Tera switch allocator leverages the fact that an input port which receives at least one output grant in the OA stage is guaranteed to transfer flits and hence should be excluded from resending requests to subsequent allocations to later iterations of iSLIP. Thus, the Tiny Tera switch allocator can start the RQ stage of the next iteration without waiting for the IA/CU stage of the previous iteration to complete. Fig 5.4 shows the IA/CU-to-RQ hazard being omitted (solid arrow in Fig 5.3) so that two iterations of one round complete in 5 cycles. In general, Tiny Tera can start a new round of iSLIP arbitration every $2i + 1$ cycles rather than every $3i$ cycles in the unpipelined case where i is the number of iterations per round assuming each stage of iSLIP takes 1 cycle. In contrast, my approach can start a new round every cycle.

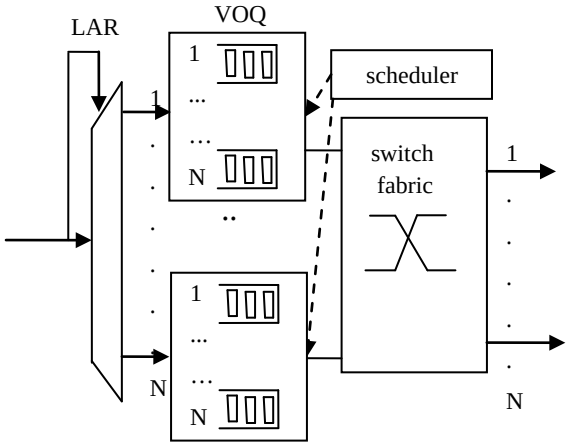

Fig 5.5 VOQ Router

Kim et al. propose using buffered crossbars in high-radix on-chip routers [78]. The buffers in the crossbar act like limited VOQs further reducing HOL blocking. The performance of their switch allocator is bounded by that of the SPAA allocator (with VOQs) because of their use of input arbitration followed by output arbitration.

5.4. apSLIP

Recall from Section 5.1 that apSLIP employs VOQ to eliminate HOL blocking combined with my two innovations (1) high-throughput pipelined iSLIP switch allocation, and (2) adaptive-effort switch allocation. I discuss both components in detail next.

5.4.1. Virtual Output Queuing in On-chip Networks

As mentioned in Section 5.1, VOQ has one fundamental operational difference vis-à-vis flow-controlled (i.e., backpressured) networks that use VCs. Essentially, VOQ requires the destination router to determine the home queue of an incoming flit because flits are placed in a virtual queue corresponding to the flit's output port. VC-based flow-controlled networks, on the other hand, require the source router to determine the home queue of the flit on the destination router. The source router allocates a VC on the destination router and tracks the occupancy of this VC when sending a flit to ensure that the destination router does not drop/overwrite any incoming flits.

apSLIP provides VOQ in a flow-controlled network by determining the virtual queue in which the incoming flit will reside at the source router instead of the destination router, using the well-known idea of look-ahead routing. Thus, look-ahead routing enables the use of VOQ in a backpressured network. The apSLIP router provides virtual queues at each input port for each output port of the router. The source router tracks the occupancy of these virtual queues through credits just like in flow-controlled networks with VCs. When sending a flit the source router uses look-ahead routing to determine the output port, and consequently the virtual queue, for which the flit is destined at the destination router. The source router then sends the flit when there is space available in the virtual queue.

The implications of using VOQ as opposed to VCs are many. Aside from eliminating HOL blocking, VOQ also simplifies the apSLIP router by removing VCs and the VCA stage from the pipeline. The primary goal of VCs is to prevent intermingling of flits of different packets. A VC allocated to a packet serves as an input queue which bids for the crossbar in the switch allocation stage. Hence a VC cannot have flits of multiple packets which may be headed in different directions. VOQ, on the other hand, guarantees that all flits in a queue, whether from single or multiple packets,, are headed in the same direction. Therefore there is no need to keep flits of different packets in a virtual queue separate. Note that while flits of different packets could intermingle in a virtual queue, the relative ordering of flits of a single packet is still maintained. Removing VCs lets us shorten the router pipeline by removing the VCA stage. The router determines the VOQ for an outgoing flit in the look-ahead routing stage as a function of the next hop address.

While VOQ improves performance, removing VCs from the network creates challenges which I address next. First, because I allow intermingling of flits from different packets, per-flit switch allocation requires that each flit must now carry address information, which results in slightly wider links, router buffers and crossbars (e.g., 4 bits per flit). While such per-flit address information is unnecessary for per-packet switch allocation and apSLIP can employ either per-flit or per-packet allocation, I assume the former because the latter imposes high overhead of larger buffers. Second, in the absence of VCs apSLIP is limited to dimension-ordered routing to avoid network deadlocks. (I still use multiple VOQs to avoid coherence deadlocks via virtual networks.) Dimension-ordered routing results in unbalanced usage of virtual queues in a router. Consider an apSLIP router having 4 input/output network ports (ignore local ports for this discussion) corresponding to the 4 directions in a mesh network (East, West, North, South). Therefore, each input port has 4 VOQs corresponding to the 4 output ports. Consider further that the network uses XY dimension-ordered routing which routes all flits along the X axis and then along the Y axis. In such a network, the East- and West-facing input ports of the router receive flits destined for all other ports of the router. Hence all virtual output queues are utilized on the East- and West-facing input ports. The North- and South-facing ports, however, receive flits destined only to the South- and North-facing

output ports, respectively. Therefore the VOQs corresponding to the East- and West-facing output ports are not utilized at all. I counter this imbalance by using non-uniform queue sizes as proposed in [79] [80]. The fact that VCs and VOQs are typically implemented as partitions of a single SRAM array simplifies expanding the highly-utilized queues at the expense of the under-utilized queues at design time.

5.4.2. VOQ Synegy with Pipelined Switch Allocation

Recall from Section 5.3.2 that there are two key RAW hazards that prevent naïve pipelining of iSLIP. I start my discussion on pipelining iSLIP with a high-level observation that the RAW hazards are meta-data hazards (hazards that affect request vectors and priority counters; not program data) that affect allocation performance and allocation fairness; the hazards do not affect correctness. As such, ignoring the RAW hazard and using stale information does not violate correctness. However, using stale information naively can degrade performance significantly. I outline the solutions for each of the two hazards. The first hazard is relatively easy to handle and the second hazard is a little more complicated.

Consider the first hazard between IA/CU (stage 3) and RQ (stage 1) in Fig 5.3, which is common to all pipelined switch allocators. The key challenge is that requests for subsequent allocations must be finalized before the outcome (i.e., grants) of prior allocations are determined. Using stale information (i.e., continuing to make switch requests for all outstanding flits), leads to potentially wasted allocations wherein an input port receives grants for flits which have previously been dispatched. I observe that VOQ and iSLIP synergistically mitigate the effects of this hazard.

At high loads and consequentially high VOQ occupancy, flits are available in the queue to avail a request grant from an output port. Even though the flit that caused that request is no longer in the queue, the VOQ organization makes it easy to find other flits that are destined for the same output port. At low loads, it is indeed possible that there may be switch grants that go unutilized. However, the switch allocator is not a bottleneck at low loads; thus, any wasted grants do not hurt performance as I show in my results.

5.4.3. Privatization of Priority Counters

The second hazard occurs between OA stage (stage 2) and the IA/CU stage (stage 3) on the per-port output port counters. Recall from Section 5.3.2 that an output port, whose grant is accepted by an input port, updates its private counter in IA/CU (stage 3). The output port then reads its private counter to send grants in the following OA (stage 2) (see Fig 5.3).

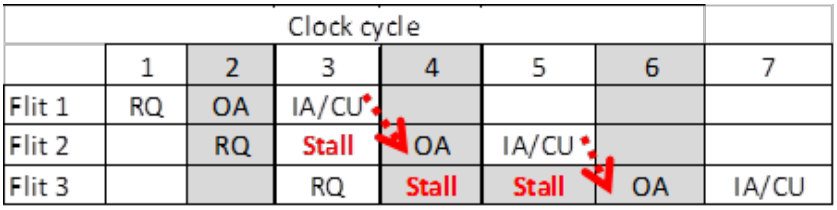

Fig 5.6Stalling to avoid pipeline hazards

A naïve solution would be to stall the pipeline stages to eliminate the hazard as shown in Fig 5.6. Unfortunately, such a solution would halve the throughput as it achieves matches only every other cycle. Instead of stalling, another naïve solution would be to use stale metadata which results in the output port in stage 2 reading an outdated counter value as shown in Fig 5.3. Unfortunately, this choice causes serious performance pathology. Specifically, reading the outdated counter value results in the output port nominating the same input port twice (i.e., two reads of the same counter value before an update) In the meantime, the input port accepts grants based on its up-to-date private counter which is read and then updated in stage 3 (Section 5.3.2). The slow-moving output port counter when coupled with the input port counter (moving at the regular rate) results in unfairness and significantly degraded performance.

The key to iSLIP's successful matching is keeping the private counters of input ports and output ports desynchronized with respect to those of the other input and output ports, respectively. Consider a scenario where two output ports keep sending grants to the same input ports because their private counters keep synchronizing. The input grants can

choose only one output port and hence the other would be wasted. Instead, by moving at the correct rate, the counters stay desynchronized.

I need to resolve the hazard between stage 3 (of flit 1) and stage 2 (of flit 2) while ensuring that both input port and output port counters move at the correct rate. My solution ignores the RAW hazard and uses stale information. To prevent the resulting counter synchronization, I propose duplicating the counters (say counter set 0 and counter set 1), effectively *privatizing* them for odd/even cycles, as shown using subscripts in Fig 5.7. This counter privatization is similar to compiler variable privatization.

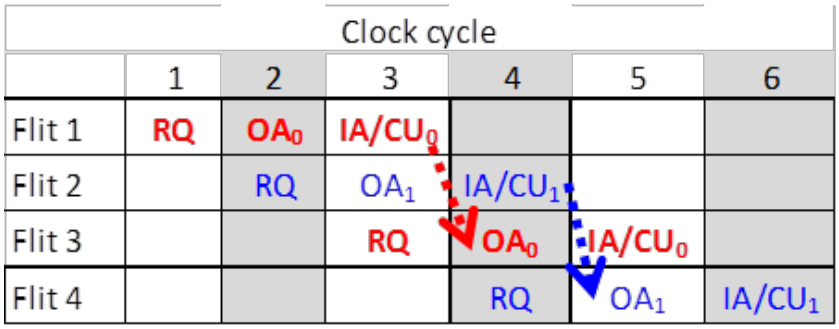

Fig 5.7 Privatisation with duplicate counters

At a high-level, my solution is equivalent to operating two independent, hazard-free allocators, each of which guarantees fairness. At a low-level, I do not actually duplicate the allocators. Rather, I *privatize* the per-port priority counters for odd and even cycles. Because of such privatization, I completely eliminate the hazard between the IA/CU and OA stages of consecutive allocations. Effectively, each allocation uses stale information from two allocations ago. Consecutive writes and reads to the same set of output port counters are now separated by two cycles (see flit 1 and flit 3 in Fig 5.7) which ensures that the updated counters are available at the end of cycle 3 before they are read in cycle 4. Further, because of the absence of races, the corresponding input and output counters are incremented at the correct rate (i.e., exactly one read per update). I have empirically examined non-uniform arrival patterns other than the example in Fig 5.7 and confirmed that my privatization is equivalent to unpipelined iSLIP.

5.4.4. Multi-Iterative and Adaptive pSLIP

It is straightforward to extend the iSLIP pipeline to create a multi-iterative iSLIP switch allocator with increased matching capability. Recall from Section 5.1 that iSLIP is an iterative, maximal-matching allocator that can achieve better matching by expending additional matching effort in the form of additional iterations. While high-load applications would benefit from the resultant higher throughput, low-load applications would lose performance due to the deeper pipeline's increased latency and higher chances of superfluous-grant mis-speculation.

To increase throughput at high loads without hurting latency at low loads, I propose an adaptive-effort iSLIP. Every router employs a six-stage, two-iteration switch allocator pipeline. However, each router independently determines whether to run a single or dual-iterative switch allocator based on network load determined by queue occupancy. At low queue occupancy, the switch allocator provides its matches at the end of the first iteration yielding a three-stage pipeline. When queue occupancy crosses a threshold, the switch allocator runs two iterations corresponding to six stages (my threshold is that half the queues have more than 6 flits each). I use hysteresis to avoid frequent changes to the pipeline depth, reverting from two to one iteration only after the queues are empty (I use a hysteresis count of 4). While changing the depth of most pipelines at runtime is usually near impossible, the iSLIP pipeline is unique in that the iterations are identical. Therefore, the pipeline may be terminated at the end of any iteration. I do not examine implementing more than two iterations as my results show diminishing returns for pipelines longer than two iterations.

In [77], the authors target pipelining instruction issue in out-of-order processors which also poses a RAW hazard problem of issuing dependent instructions back-to-back. The authors propose to have grand-parent instructions wakeup grand-child instructions instead of parent instructions waking up child instructions. My odd-even is similar in spirit. However, they do not prevent overbooking whereas my counter-privatization does.

5.5. Methodology

I use two simulators: the full-system GEMS [21] with Garnet [54] on top of Simics [20] for a 3x3 network (9 nodes) using out-of-order-issue, SMT cores and detailed

memory system models (larger systems proved infeasible due to long simulation times) and a trace-driven network-only simulator using Garnet for an 8x8 network (64 nodes). While the latter can cover larger systems, the former shows the feedback effect of the network on execution time, not shown by the latter, albeit for smaller systems.

I compare apSLIP with VoQ with several switch allocators (Section 5.2): speculative designs [46] [84], SMART [85], TS-router [88], and per-packet with and without packet chaining [83]. To cover the various speculative designs which reduce router latency, I assume an idealized pipelined SPAA-based two-cycle router at all loads which uses virtual channels (my 2-cycle baseline). The first stage overlaps look-ahead routing with ideal single-cycle VC allocation with perfect speculation. The second stage performs both the local and global arbitration phase of the SPAA switch allocator. I also show an idealized, one-cycle SPAA-based baseline router at all loads though the speculative designs achieve low latency only at low loads and incur full latency at high loads (e.g., 4 cycles). The baseline routers use 8 VCs per virtual network each with 8-flit buffers where each flit is 32 bits (justified in Section 5.2) and dimension-ordered routing. Adding more VCs did not yield any significant improvement.

I build SMART on the 1-cycle baseline and the other previous schemes, TS-router, per-packet, and packet chaining, on the 2-cycle baseline. I include several idealizations for each of these schemes. For SMART, I assume that the router latency and router set-up are zero cycles for route segments without any contention so that the only latencies are 1-cycle inter-router wire delay (the SMART paper assumes 9 routers traversed in one 1-GHz cycle) and 1-cycle router delay with contention. For TS-router, per-packet, and packet chaining, I assume an ideal one-cycle switch allocation (2 cycles total router latency). The per-packet switch allocator uses large, packet-sized 18-flit buffers while packet chaining uses the large buffers and an idealized, collision-free, duplicate allocator that finds the best chaining candidate among all the packets in the switch. The baselines and all other schemes run at 2.8 GHz (same as core frequency) and have a 1-cycle wire delay in the links beyond the router latency.

Table 5.1Workload parameters

Name	Run	Input	Warmup	Inj Rate
Commercial Workloads – Trace				
Apache	200 million cycle trace	20,000 files 45,000 clients 25 ms think time	20,000 *tx cache* *2 million system*	
SPECjbb		90 warehouses	50,000 *tx cache* *1 million system*	
Online Transaction Processing (OLTP)		25000 warehouses 300 connections	5000 *tx cache* *0.1 million system*	
Commercial Workloads – Full System				
Apache	600 *tx*	20,000 files 45,000 clients 25 ms think time	20,000 *tx cache* *2 million system*	0.78
SPECjbb	3000 *tx*	90 warehouses	50,000 *tx cache* *1 million system*	0.77
Online Transaction Processing (OLTP)	50 *tx*	25000 warehouses 300 connections	5000 *tx cache* *0.1 million system*	0.68
SPLASH-2 Workloads – Trace				
Ocean	200 million cycle trace	258x258 grid		
Barnes		16384 particles		
Water-nsquared		512 molecules		
SPLASH-2 Workloads – Full System				
Ocean	Full	34x34 grid		0.19
Barnes	Full	512 particles		0.1
Water-nsquared	1 time step	64 molecules		0.09

My apSLIP implementation models unevenly partitioned virtual output queues at each input port, sharing a pool of 64-flit buffers. Hence apSLIP has the same number of buffers as the baseline. Each input port has as many VOQs as there are output ports. apSLIP's adaptive pipeline varies between four stage and seven stages. The first stage includes look-ahead routing and VOQ allocation followed by one and two iterations of the iSLIP algorithm's RQ, OA, IA/CU stages at low and high loads, respectively (Section 5.3.2). apSLIP uses 4 extra bits per flit for address (Section 5.4.1).

Table 5.2 System configuration

System	1 chip 9 cores
Network	3x3 mesh, each node is a core and an L2 cache bank; flit width is 32 bit, 2 control virtual networks and 1 data network (8 + 8 + 8= 24 VCs) with 8-flit-deep buffers; 2-cycle link latency
Cores	4-way SMT, 4-issue out-of-order with 40-entry instruction window
Private L1 Caches	Split I & D, each 64KB, 4-way set associative, 64-byte blocks, 2-cycle latency, 16 MSHRs
Shared L2 Cache	Unified 18MB with 9 banks, 16-way set associative with LRU, 12-cycle latency, 16 MSHRs
Memory	8 GB DRAM, 250-cycle off-chip access time, 2 DIMMs per channel, 2 ranks per DIMM, 8 banks per rank, 32 bank queue entries

Full-System Simulation: The simulated system has 9 out-of-order-issue, 4-way SMT cores; Table 5.2 summarizes the system's key parameters. The benchmarks include three high-load/commercial and three low-load/scientific multi-threaded applications (Table 5.1 column "Inj Rate" gives the load in terms of per-cycle, per-core injection rate). I run the commercial benchmarks for a fixed number of transactions after adequate cache warm-up. I scale scientific workloads from SPLASH-II [61] to run to completion.

Trace-Driven Network-only Simulation: I gather traces from full-system simulations of 64 in-order-issue 2-way SMT cores to run on an 8x8 2-D mesh network simulator with 64 nodes. I scale the execution rate of each individual trace to match the per-thread instructions per cycle of an out-of-order-issue 2-way SMT core. I then apply a constant scaling factor to the scale down the execution rate of all traces to compensate for the smaller bisection bandwidth of the 8x8 network to avoid network saturation for my baseline. Note that such downscaling is conservative as it helps the baseline avoid latency explosion associated with network saturation. I run the same set of benchmarks as I ran for full system simulation. I scale the SPLASH –II workloads to run on a 64 node system. (See Table 5.1). I measure average per-flit latency after adequate network warm-up.

Apart from commercial applications and SPLASH-II benchmarks, I also show several synthetic workloads with both data and control packets.

Circuit Analysis: To analyze apSLIP's circuit delays, I model the two key stages (OA and IA/CU) of apSLIP and SPAA in Verilog, verify the functionality using MentorGraphics's ModelSim, and synthesize the model in 45 nm technology using Synopsys's Design compiler. I do not model the third lightweight stage which includes only wire delays for forwarding requests.

5.6. Results

I start with the main comparison of apSLIP+VOQ with several previous switch allocators for a 3x3 network (full-system) and an 8x8 network (trace-driven network-only) on commercial and scientific workloads. I then isolate the impact of VOQ and adaptive allocation. Next, I analyze apSLIP's performance using synthetic workloads. Finally, I present circuit-level analysis of the apSLIP+VOQ router.

5.6.1. Performance

Fig 5.8 plots application performance (full-system) for a 3x3 network connecting 9 out-of-order-issue 4-way SMT cores. The Y axis shows performance for my 1-cycle SPAA-based baseline, idealized SMART on top of the 1-cycle baseline, idealized TS-router on top of the 2-cycle baseline, per-packet switch allocation without and with packet chaining on top of the 2-cycle baseline normalized to that of my 2-cycle SPAA-based baseline. The X axis shows my commercial (high load) and SPLASH-II (low load) benchmarks in the order of decreasing load (Table 5.1).

For the high-load benchmarks, the 1-cycle baseline, SMART and TS-router do not perform better than the 2-cycle baseline. The 1-cycle baseline and SMART are limited by SPAA's poor matching power. The SMART paper shows higher speedups due to ultra low loads caused by wide 128-bit flits (more than 500-bit link width) whereas the load is higher at QPI-like 32-bit link widths making SMART's favorable case of contention-free route segments uncommon. Recall from Section 2, TS-router's anticipatory evacuation is unable to adequately evacuate the input port queues at high loads and is thus unable to improve scheduling. The TS-router paper shows around 2% improvement over packet

chaining which I do not see because the TS-router paper uses buffers that are smaller than a packet which impedes packet chaining (I use packet-sized buffers for packet chaining). Per-packet allocation without and with packet chaining fare better than the previous schemes; packet chaining achieves 10% mean speedup which is in line with the packet chaining paper. Note that packet chaining has ideal choice of which candidates to chain. Still, per-packet allocation is limited by control packets and the HOL blocking in VCs; and packet chaining alleviates but does not eliminate the control packet problem nor the HOL blocking in VCs. Finally, apSLIP improves performance significantly by employing VOQ to remove HOL blocking and iSLIP to achieve high-quality allocation. apSLIP improves the high-load commercial workloads by nearly 20%. Recall that while packet chaining needs large buffers and duplicate allocators and that my implementation uses an idealized, collision-free packet chaining allocator (Section 5.5), apSLIP performs better without incurring such overheads. For the low-load SPLASH-II benchmarks, the network has little impact on overall performance and hence all the schemes perform similarly (apSLIP is within 5%).

Fig 5.9 plots the network latency (trace-driven network-only) for an 8x8 network. The Y axis shows 1/network latency for 1-cycle baseline, idealized SMART, idealized TS-router, per-packet switch allocation without and with packet chaining normalized to that of my 2-cycle baseline (higher is better). The X axis shows my commercial and SPLASH-II benchmarks in the order of decreasing load (Table 5.1). The trends from the full-system simulations (Fig 5.8) hold though the two graphs plot different metrics and benchmarks, and therefore should not be compared directly. Unlike the full system run though, the SPLASH-II benchmarks with 64 threads offer a high enough network load to be affected by choice of switch allocator. While the 1-cycle baseline does not improve latency as in Fig 5.8, the SPLASH-II benchmarks benefit from the TS-router because of favorable network load and the SMART router which reduces the longer latency of the 8x8 network as compared to the 3x3 network in Fig 5.8. Per-packet allocation without and with packet chaining perform moderately well as in Fig 5.8. apSLIP improves latency by 34%, performing significantly better than the others.

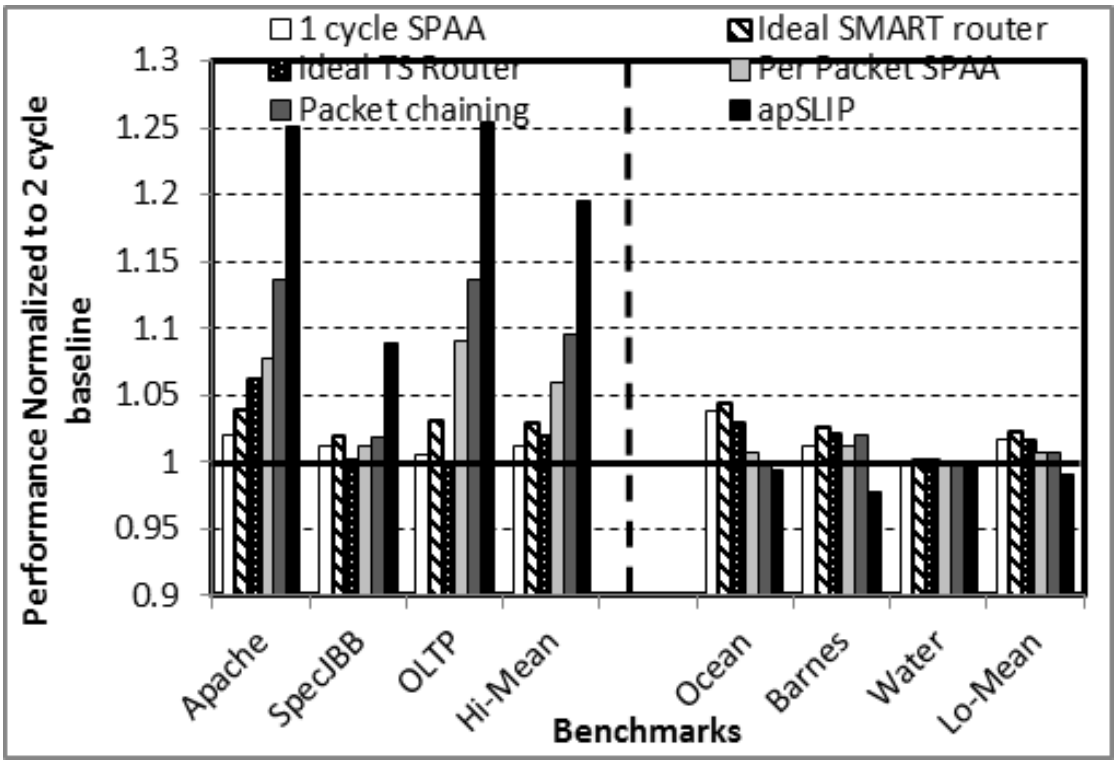

Fig 5.8 System performance improvement

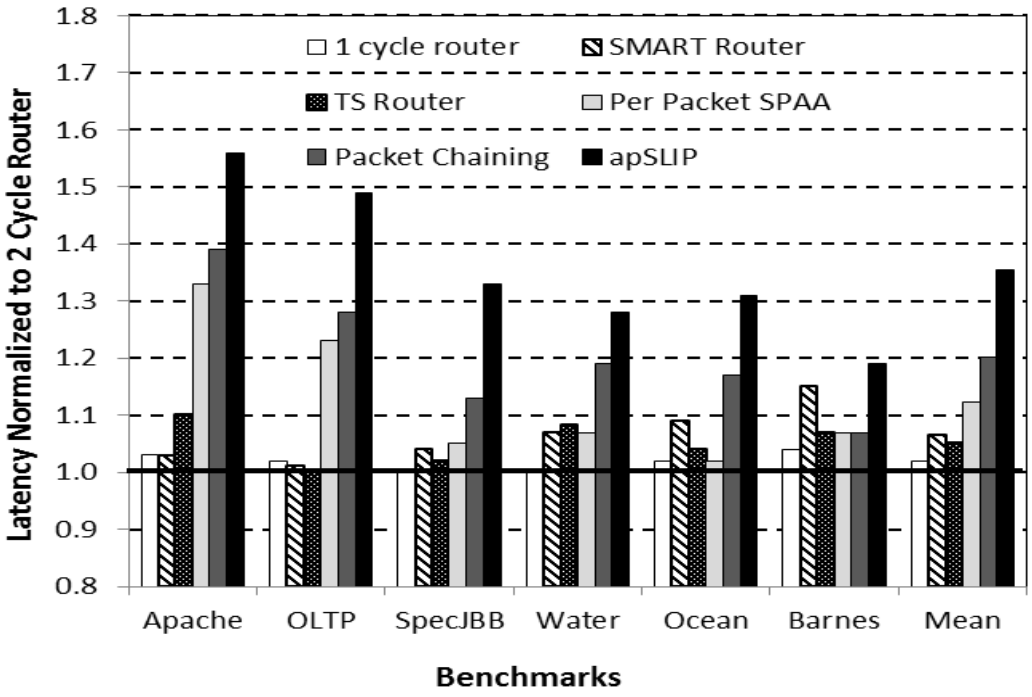

Fig 5.9 Network latency improvement

5.6.2. Performance breakdown

The apSLIP scheme combines VOQ, pipelined iSLIP, and adaptive pipelining (3- or 6-stage pipeline based on network load). I now isolate the impact of these components. I isolate the impact of VOQ's elimination of HOL blocking from apSLIP in Fig 5.10 and of apSLIP's adaptive pipelining in Fig 5.11. I do not isolate the pipelining part of apSLIP because unpipelined iSLIP can generate a match only once every three cycles as opposed to every cycle which would incur severe performance loss. I use trace-driven simulation for Fig 5.10 to evaluate the impact of VOQ on a large 8x8 network and full-system simulations for Fig 5.11 to provide realistic injection rates to study the impacts of adaptivity. The full system simulation is feasible only for a small 3x3 network.

In Fig 5.10, I isolate VOQ's impact by comparing SPAA with VOQ and apSLIP with VOQ (i.e., my full scheme) in an 8x8 network running my commercial and SPLASH-II benchmarks. The Y axis shows 1/network latency for SPAA combined with VOQ (2-cycle router latency like the baseline) and apSLIP (4- to 7-cycle router latency)

normalized to that of my 2-cycle baseline (SPAA with VC). VOQ with SPAA improves over VC with SPAA (my baseline) by 12% due to VOQ's removal of HOL blocking. apSLIP adds another 22% to VOQ's improvements for a total of 34% (i.e., both components of apSLIP give good benefits). Some benchmarks (ocean, barnes and to some extent apache) do not incur HOL blocking and hence do not benefit from VOQ. Other benchmarks such as OLTP, SpecJBB and water exhibit HOL blocking and hence show significant improvement with just VOQ.

In Fig 5.11, I isolate apSLIP's adaptivity by comparing apSLIP against pipelined iSLIP with static 3 and 6 stages in a 3x3 network running my commercial and SPLASH-II workloads. The Y axis shows full-system performance for 3-stage, 6-stage, and adaptive pipelined iSLIP normalized to that of my 2-cycle baseline. While the high-load commercial workloads prefer the 6-stage pipeline's higher bandwidth over the 3-stage pipeline's lower latency, the low-load scientific workloads reverse this preference. Being adaptive, apSLIP performs better than or close to the better of the two static pipelines across all loads.

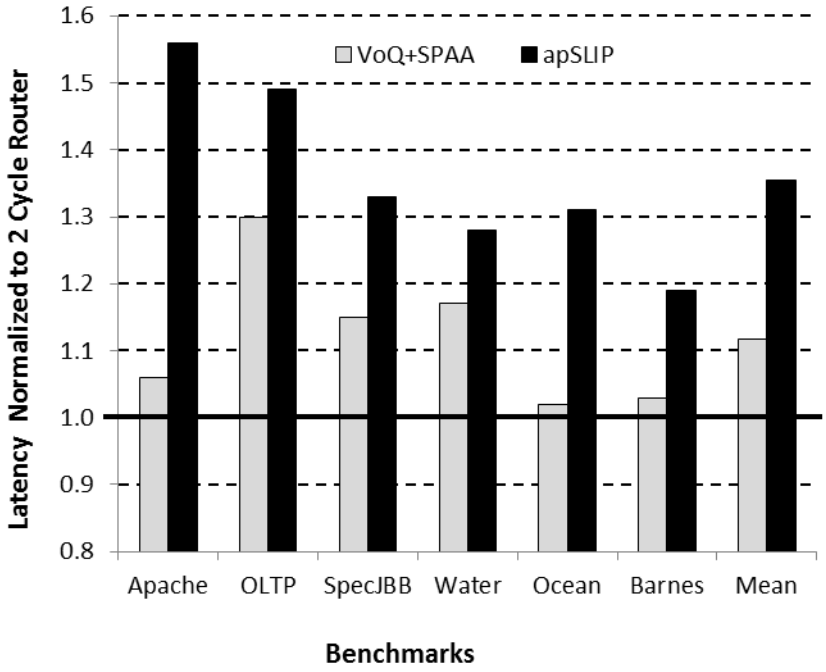

Fig 5.10 Impact of VOQ on network latency

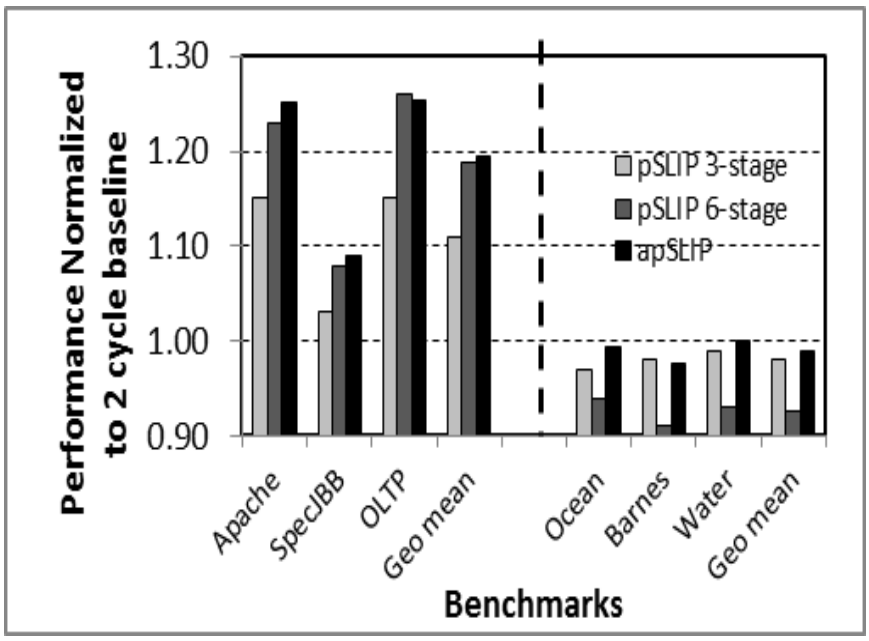

Fig 5.11 Impact of adaptivity on system performance

5.6.3. Synthetic Workloads

To better understand apSLIP's performance, I run synthetic workloads of traffic patterns, namely nearest neighbor, uniform random, transpose, and bit complement (in the order of increasing difficulty). In Fig 5.12-5.15, I plot the average network latency (Y axis) versus the injection rate in flits per node per cycle (X axis) for SPAA with per-packet switch allocation, SPAA with packet-chaining, and apSLIP for each of the said traffic patterns. In the nearest neighbor pattern (Fig 5.12), each node sends packets to its immediate right neighbor. There is no contention and all three schemes saturate near injection rate of 1. The uniform random pattern has contention without hot spots and therefore emphasizes switch allocation. In this pattern (Fig 5.13), per-packet saturates first followed by packet chaining and apSLIP which performs best. In the bit complement and transpose patterns (Fig 5.14 and Fig 5.15), which stress the network bisection, all three schemes saturate near injection rate of 0.85. These results show that apSLIP is robust across traffic patterns and performs better when switch allocation matters.

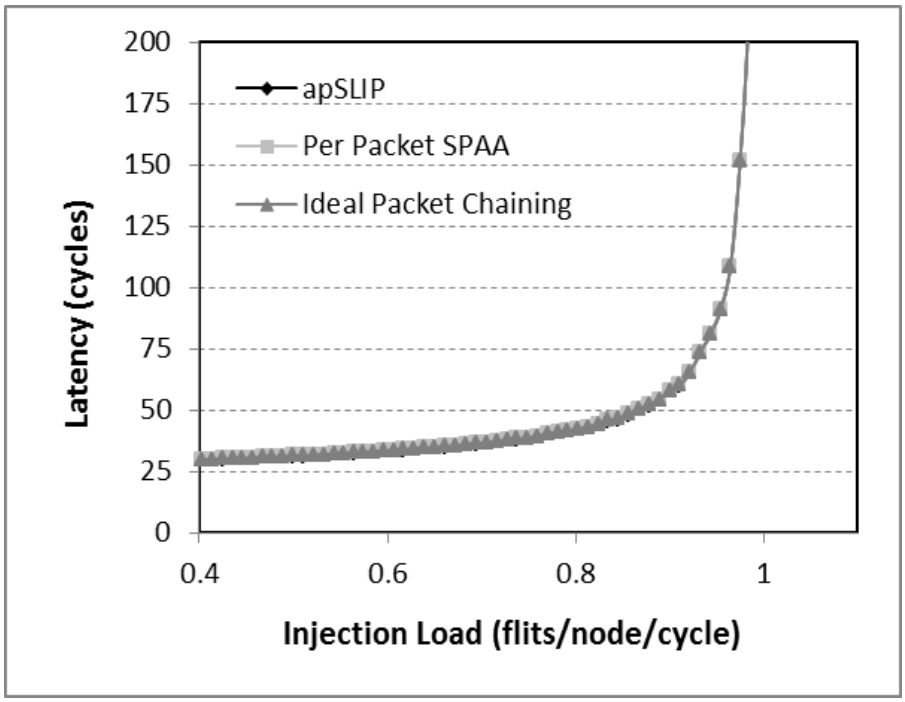

Fig 5.12 Latency vs Injection Load - Nearest Neighbor Pattern

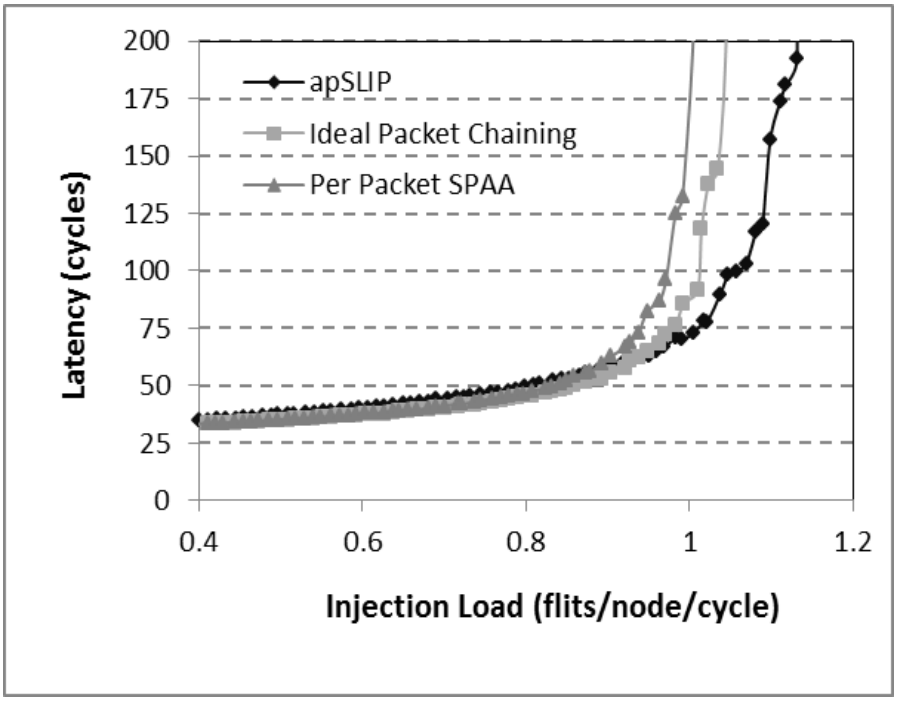

Fig 5.13 Latency vs Injection Load - Uniform Random Pattern

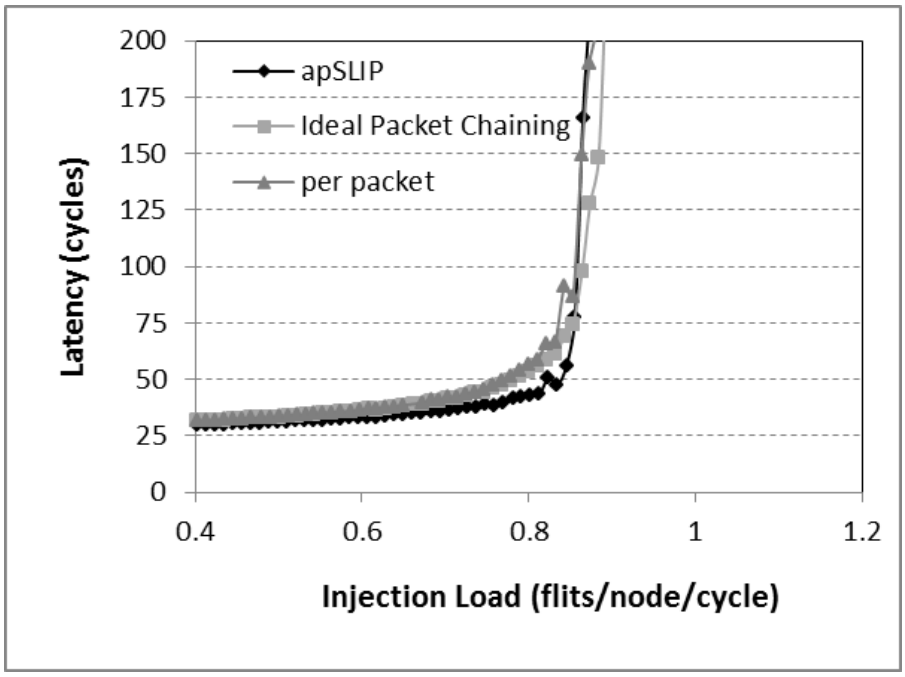

Fig 5.14 Latency vs Injection Load - Bit Complement Pattern

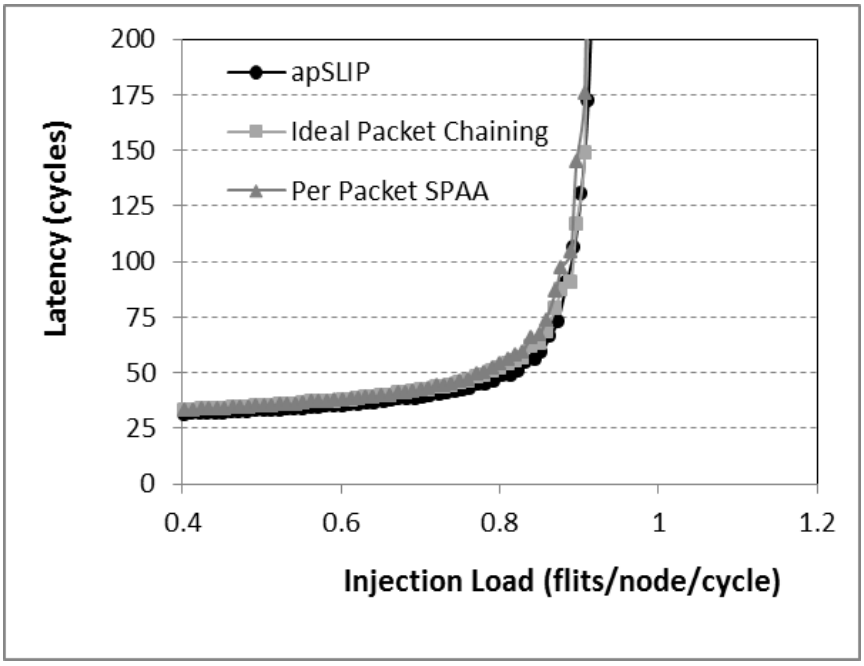

Fig 5.15 Latency vs Injection Load - Transpose Pattern

5.6.4. Circuit Analysis

Comparing SPAA and apSLIP's circuit-level implementations, the input and output arbitrations are qualitatively identical in SPAA and apSLIP. While only the counter updates differ, they are off the critical path. Accordingly, the synthesized models show little difference in clock speeds between SPAA and apSLIP with SPAA allocator at 7.6 FO4 (243 ps) and apSLIP allocator at 7.8 FO4 (249 ps) whereas the input buffer write is the critical path at 8.7 FO4 (278 ps).

5.7. Conclusion

Switch allocation and queuing discipline has a first-order impact on network performance; and hence on overall system performance. Quality of switch allocation and clock-speed impose opposing constraints: Dependencies in sophisticated switch allocation algorithms such as iSLIP make pipelining at fast clocks hard. On the other hand, simpler, pipelineable algorithms which are amenable to fast clocks degrade throughput.

This chapter proposes apSLIP, a high-performance, adaptive-effort, pipelined switch allocator. apSLIP uses three novel ideas to pipeline iSLIP. First, I break the request-grant RAW hazard by leveraging VOQ which easily allows another flit to avail a superfluous grant. Second, I untangle double-booking problem arising from priority counter RAW hazard by privatizing priority counters. Finally, I use adaptive effort switch allocation to achieve high-bandwidth at high loads (via a deeper pipeline) and low latency at low loads (via a shallower pipeline). Simulations reveal that apSLIP improves performance by 20% on average over an aggressive 2-cycle router baseline for high-load benchmarks without affecting the low-load benchmarks. apSLIP's high bandwidth and low latency are important for on-chip networks to keep up with the ever-growing core counts of multicores.

6. CONCLUSION

Transactional memory and multi-hop interconnection networks make inefficient use of resources and degrade in performance. TM designs require large amount of memory hierarchy space to store metastate and lose performance because of current conflict resolution policies. Similarly on-chip networks require a significant fraction of total processor energy, and suffer from performance bottlenecks such as head-of-line blocking and poor switch arbitration. For my dissertation I make common case observations to reduce resource requirements and improve performance for TM designs and multi-hop interconnection networks.

Transactional memory is a promising alternative to lock based parallel programming, but recent HTMs (e.g., TokenTM, VTM, OneTM-concurrent) employ per block metastate which can result in considerable overhead. I propose LiteTM which cuts the overhead by decoupling the detection of conflicts (done in hardware) from the identification of conflicting transactions (done in software using transactional logs, in the uncommon case). Experiments show that LiteTM reduces TokenTM's state overhead by about 87% while performing within 4%, on average, and 10%, in the worst case, of TokenTM. By reducing transactional state overhead while maintaining performance, LiteTM lowers the barrier for adoption of HTMs in real products.

Conflict resolution policies significantly impact TM performance. Previous policies degrade performance in the presence of contention by limiting concurrency or by incurring late aborts. Further, none of the policies avoid aborts due to cyclic dependencies. I stipulated that conflict resolution policies should increase concurrency while avoiding cyclic aborts. To that end, I proposed Wait-n-GoTM (WnGTM) based on the key observation that most cyclic dependencies (and hence aborts) are caused by threads interleaving accesses to a few heavily-read-write-shared delinquent data cache

blocks. The Wait-n-Go (WnG) policy employs hardware prediction to serialize sections of inflight transactions to avoid many cyclic abort. I showed that WnGTM achieves, on average, 46% and 28% speedups over TokenTM for the higher-contention benchmarks and all the benchmarks, respectively, with low-contention benchmarks remaining unchanged.

As the microprocessor industry packs more cores into a chip, multi-hop interconnection networks are likely to be used as the on-chip communication fabric. Network flow-control mechanisms have a first order impact on the performance and energy of networks. Two widely-studied flow-control mechanisms – credit-based backpressured flow control and backpressureless deflection flow control – have their own particular network load "sweet spots" where they operate well but incur significant performance/energy penalties at loads outside their sweet spots. Unfortunately, workload characteristics are not limited to the "sweet spot" region of any single flow control mechanism.

I propose Adaptive Flow Control (AFC) – a robust flow control mechanism with a wide sweet spot that spans high and low loads. AFC routers operate in backpressureless mode at low loads and as backpressured routers at high loads. Experimental results show that AFC's performance and energy are close to those of the better of backpressured and backpressureless routers. As the number of cores continues to scale, and as the mix of applications grows more diverse, AFC's performance and energy robustness will be increasingly important.

Switch allocation and queuing discipline have a first-order impact on network performance; and hence on overall system performance. Unfortunately, sophisticated switch allocation mechanisms such as iSLIP require a slow clock while simpler allocation algorithms which are pipelineable (and hence amenable to fast clocks) degrade throughput. I propose a high-performance, adaptive-effort, pipelined switch allocator – apSLIP. apSLIP improves performance by 30% on average on an 8x8 network for a suite of workloads including Parsec benchmarks.

In conclusion, multi-cores have replaced uniprocessors, therefore challenges related to programmability and scalability of multi-cores are of paramount importance.

Transactional memory is an alternative to error prone lock-based programming , while multi-hop interconnection networks hold the promise of scalable inter-core communication. In my dissertation I identify and resolve some key challenges related to the resource usage and performance of both transactional memory and multi-hop interconnection networks.

7. FUTURE WORK

While my current work on multi-hop networks centers on the communication fabric, there are several performance bottlenecks and opportunities for reducing energy consumption in the network interface. Depending on the scale of the network, the network interface hardware and software (if any) can introduce variable delay in the delivery of traffic onto the communication fabric. For larger scale networks, such as external networks for high performance computing clusters, I believe there is potential in innovations in the structure of the network interface hardware, and lighter weight communication protocols to reduce communication overheads as well as shorten the height of the communication stack. While reducing the latency of the communication stack has a first order impact on throughput, reducing the variance in the latency as may be imposed by interrupt based handling of network traffic leads to a load balanced network which improves overall throughput.

As a future extension to my dissertation I plan to explore innovations in the network interface hardware to support lighter weight communication protocols specialized for high performance computing.

LIST OF REFERENCES

LIST OF REFERENCES

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In PLDI ’06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pages 26–37. ACM, 2006.

[2] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In HPCA ’03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture, page 7. IEEE Computer Society, 2003.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture, pages 316–327. Feb 2005.

[4] J. Archibald and J. L. Baer. An economical solution to the cache coherence problem. In Proceedings of the 11th Annual International Symposium on Computer Architecture. 1984.

[5] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In Proceedings of the 35th Annual International Symposium on Computer Architecture. June 2008.

[6] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. SIGARCH Comput. Archit. News, 35(2):24–34, 2007.

[7] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. Tokentm: Efficient execution of large transactions with hardware trans- actional memory. In Proceedings of the 35th Annual International Symposium on Computer Architecture. Jun 2008.

[8] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In ISCA ’07: Proceedings of the 34th annual international symposium on Computer architecture, pages 81–91. ACM, 2007.

[9] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In IISWC'08: Proceedings of The IEEE International Symposium on Workload Characterization, September 2008.

[10] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In Proceedings of the 34th Annual International Symposium on Computer Architecture. Jun 2007.

[11] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In ISCA '06: Proceed- ings of the 33rd annual International Symposium on Computer Architecture, pages 227–238. IEEE Computer Society, 2006.

[12] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. SIGPLAN Not. 41(11):347–358, 2006.

[13] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. SIGOPS Oper. Syst. Rev., 40(5):371–381, 2006.

[14] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 336–346, 2006.

[15] J. Gray. The transaction concept: Virtues and limitations. In Seventh International Conference on Very Large Data Bases, pages 144–154. Sep 1981.

[16] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. IEEE Micro, 24(6), Nov-Dec 2004.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture, pages 289–300. May 1993.

[18] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In ASPLOS '09: Proceeding of the 14th inter- national conference on Architectural support for programming languages and operating systems, pages 145–156. ACM, 2009.

[19] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In Proceedings of Symposium on Principles and Practice of Parallel Programming, Mar 2006.

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. IEEE Computer, 35(2):50–58, Feb. 2002.

[21] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput. Archit. News, 33(4):92–99, 2005.

[22] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In Proceedings of the 12th International Symposium on High-Performance Computer Architecture, pages 254–265. Feb 2006.

[23] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logTM. SIGPLAN Not., 41(11):359–370, 2006.

[24] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In Proceedings of the 32nd Annual International Symposium on Computer Architecture, pages 494–505. IEEE Computer Society, Jun 2005.

[25] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 123–133. IEEE Computer Society, 2007.

[26] N. Shavit and D. Touitou. Software transactional memory. In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 204–213. Aug 1995.

[27] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture, pages 139–150. IEEE Computer Society, 2008.

[28] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA). Feb 2007.

[29] G. Blake, R. G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 156-167, 2009.

[30] C. Blundell, A. Raghavan, and M. M. Martin. RETCON: transactional repair without replay. In Proceedings of the 37th Annual International Symposium on Computer Architecture, pages 258--269, 2010.

[31] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 extension for lock-free data structures and transactional memory. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pages 39--50, 2010.

[32] C. Click. Azul's experiences with hardware transactional memory. In Transactional Memory Workshop, 2009.

[33] Intel Corporation. Intel architecture instruction set extensions programming reference, 319433-012a edition. 2012.

[34] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 157--168, 2009.

[35] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 181--193, 1997.

[36] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In Proceedings of the 32nd Annual International Symposium on Computer Architecture, pages 494--505, 2005.

[37] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, pages 246--257, 2008.

[38] A. Sodani. Race to exascale: Opportunities and challenges. In Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture, 2011.

[39] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar. Timetraveler: exploiting acyclic races for optimizing memory race recording. In Proceedings of the 37th Annual International Symposium on Computer Architecture, pages 198--209, 2010.

[40] M. Waliullah and P. Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing, pages 1--11, 2008.

[41] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, pages 127--136, 2012.

[42] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and T. N. Vijaykumar. Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies. In Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13), 2013.

[43] H. Wang, L.-S. Peh, and S. Malik, “Power-driven design of router microarchitectures in on-chip networks,” in MICRO 36: Proc. of the 36th Annual IEEE/ACM Int’l Symp. on Microarchitecture, 2003, p.105.

[44] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, “Express virtual channels: towards the ideal interconnection fabric,” in ISCA ’07: Proc. of the 34th Annual Int’l Symp. on Computer architecture, 2007, pp. 150–161.

[45] T. Moscibroda and O. Mutlu, “A case for bufferless routing in onchip networks,” in ISCA ’09: Proc. of the 36th Annual Int’l Symp. On Computer Architecture, 2009, pp. 196–207.

[46] M. Hayenga, N. E. Jerger, and M. Lipasti, “SCARAB: a single cycle adaptive routing and bufferless network,” in MICRO 42: Proc. of the 42nd Annual IEEE/ACM Int’l Symp. on Microarchitecture, 2009, pp. 244–254.

[47] S. Konstantinidou and L. Snyder, “The chaos router: a practical application of randomization in network routing,” in SPAA ’90: Proc. of the Second Annual ACM Symp. on Parallel Algorithms and Architectures, 1990, pp. 21–30.

[48] P. Baran, “On distributed communications networks,” IEEE Trans. On Communications Systems, vol. 12, no. 1, pp. 1–9, March 1964.

[49] J. Rabaey, A. Chandrakasan, and B. Nikolic, Digital Integrated Circuits, 2nd ed. Prentice Hall Inc., 2002.

[50] W. Dally and B. Towles, Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers Inc., 2003.

[51] L.-S. Peh and W. J. Dally, “A delay model and speculative architecture for pipelined routers,” in HPCA ’01: Proc. of the 7th Int’l Symp. On High-Performance Computer Architecture, 2001, p. 255.

[52] G. Michelogiannakis, D. Sanchez, W. J. Dally, and C. Kozyrakis, “Evaluating bufferless flow control for on-chip networks,” in Proc. of the Fourth ACM/IEEE Int’l Symp. on Networks-on-Chip, 2010, pp. 9–16.

[53] R. M. Andrew, A. West, and S. Moore, “Low-latency virtual-channel routers for on-chip networks,” in In Proc. of the 31st Annual Int’l Symp. on Computer Architecture, 2004, pp. 188–197.

[54] L.-S. Peh, N. Agarwal, N. Jha, and T. Krishna, “Garnet: A detailed on-chip network model inside a full-system simulator,” in Int’l Symp. on Performance Analysis of Systems and Software (ISPASS), 2009.

[55] Syed Ali Raza Jafri, Yu-Ju Hong, Mithuna Thottethodi, and T. N. Vijaykumar. Adaptive Flow Control for Robust Performance and Energy. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43), 2010.

[56] N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar, “Next generation Intel micro-architecture (Nehalem) clocking architecture,” in IEEE Symp. on VLSI Circuits, 2008, pp. 62–63.

[57] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded Sparc processor,” IEEE Micro, vol. 25, no. 2, pp. 21 29, 2005.

[58] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, “The AMD Opteron processor for multiprocessor servers,” IEEE Micro, vol. 23, no. 2, pp. 66–76, 2003.

[59] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson et al., “An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS,” in IEEE ISSCC Dig. Tech. Papers, 2007, pp. 98 – 99.

[60] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, “Orion: a power performance simulator for interconnection networks,” in MICRO 35: Proc. of the 35th Annual ACM/IEEE Int’l Symp. on Microarchitecture, 2002, pp. 294–305.

[61] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: characterization and methodological considerations,” in ISCA ’95: Proc. of the 22nd Annual Int’l Symp. on Computer architecture, 1995, pp. 24–36.

[62] B. J. Smith, “A pipelined, shared resource MIMD computer,” Advanced computer architecture, pp. 39–41, 1986.

[63] B. J. Smith, “Architecture and applications of the HEP multiprocessor computer system,” Readings in computer architecture, pp. 342–349, 2000.

[64] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, “The Tera computer system,” in ICS ’90: Proc. of the 4th Int’l Conf. on Supercomputing, 1990, pp. 1–6.

[65] J. Bannister, F. Borgonovo, L. Fratta, and M. Gerla, “A performance model of deflection routing in multibuffer networks with nonuniform traffic,” IEEE/ACM Trans. Netw., vol. 3, no. 5, pp. 509–520, 1995.

[66] N. Maxemchuk, “Comparison of deflection and store-and-forward techniques in the Manhattan Street and shuffle-exchange networks,” in INFOCOM ’89. Proc. of the Eighth Annual Joint conf. of the IEEE Computer and Communications Societies. Technology: Emerging or Converging, 1989, pp. 800–809 vol.3.

[67] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, “Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip,” in DATE ’04: Proc. of the conf. on Design, automation and test in Europe, 2004, p. 20890.

[68] Z. Lu, M. Zhong, and A. Jantsch, “Evaluation of on-chip networks using deflection routing,” in GLSVLSI ’06: Proc. of the 16th ACM Great Lakes Symp. on VLSI, 2006, pp. 296–301.

[69] C. G´omez, M. E. G´omez, P. L´opez, and J. Duato, “Reducing packet dropping in a bufferless NoC,” in Euro-Par ’08: Proc. of the 14th Int’l Euro-Par conf. on Parallel Processing, 2008, pp. 899–909.

[70] X. Chen and L.-S. Peh, “Leakage power modeling and optimization in interconnection networks,” in ISLPED ’03: Proc. of the 2003 Int’l Symp. on Low power electronics and design, 2003, pp. 90–95.

[71] M. Karol, M. Hluchyj and S. Morgan, "Input versus output queuing on a space division switch," IEEE Trans. Communications, vol. 35, no. 12, pp. 1347-1356, 1987.

[72] M. J. Karol, K. Y. Eng and H. Obara, "Improving the performance of input-queued ATM packet switches," in Proc. of INFOCOM, 1992.

[73] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," IEEE/ACM Trans. Networks, vol. 7, no. 2, pp. 188-201, 1999.

[74] S. S. Mukherjee, et al., "A comparative study of arbitration algorithms for the Alpha 21364 pipelined router," in Proc. of ASPLOS-X, 2002.

[75] W. J. Dally, "Virtual-channel flow control," in Proc. of the 17th ISCA, 1990.

[76] Y. Tamir and H. C. Chi, "Symmetric Crossbar Arbiters for VLSI Communication Switches," IEEE Trans. Parallel. Distrib. Syst., vol. 4, no. 1, pp. 13-27, 1993.

[77] J. Stark, M. D. Brown and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," in Proc. of the 33rd MICRO 33, 2000.

[78] J. Kim, W. J. Dally, B. Towles and A. K. Gupta, "Microarchitecture of a High-Radix Router," in Proc. of the 32nd ISCA, 2005.

[79] Y. Tamir and G. L. Frazier, ""High-performance multiqueue buffers for VLSI communication switches," in Proc. of the 15thISCA, 1988.

[80] C. A. Nicopoulos, et al., "ViChaR: A Dynamic Virtual Channel Regulator for Network-on-Chip Routers," in Proc. of the 39th MICRO, 2006.

[81] T. E. Anderson, et al., "High-speed switch-scheduling for local area networks," ACM Trans. Comput. Syst., vol. 11, no. 4, pp. 319-352, 1993.

[82] P. Gupta and N. McKeown, "Designing and Implementing a Fast Crossbar Scheduler," IEEE Micro, vol. 19, no. 1, pp. 20-28, 1999.

[83] George Michelogiannakis, Nan Jiang, Daniel Becker, and William Dally, "Packet chaining: efficient single-cycle allocation for on-chip networks," in Proc. of MICRO-44, 2011.

[84] A. Kumar, et al., "A 4.6Tbits/s 3.6GHz single-cycle NoC router with a novel switch allocator in 65nm CMOS," in Proc. of the 25th ICCD 2007.

[85] Tushar Krishna, et al, "Breaking the On-Chip Latency Barrier Using SMART," in Proc. of 19th HPCA, 2013.

[86] Intel Corporation, "An Introduction to the Intel® QuickPath Interconnect," Intel Corporation, 2009.

[87] Yatin Hoskote, et al, "A 5-GHz Mesh Interconnect for a Teraflops Processor," IEEE Micro, pp. 51-61, September 2007.

[88] Yuan-Ying Chang, et al, "TS-Router: On Maximizing the Quality-of-Allocation in the On-Chip Network," in Proc. of the 19th HPCA, 2013.

[89] S. Vangal, et al., "An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS," in Proc. of the ISSCC, 2007.

APPENDIX

A. CYCLE INDUCER SECTIONS

The code shown below shows a CIST in intruder from STAMP [10]. The CIST occurs in transaction #2 and the block pointed to by queuePtr is delinquent. For brevity, we show only the lines of code (with line numbers) pertaining to the CIST.

```
-----------------------------------------------------------------
[1] BEGIN_TRANSACTION(2)
[2] TMdecoder_process(decoderPtr,...);
[3] END_TRANSACTION(2)

[1] TMdecoder_process(decoderPtr,...) {
            :
            // check for errors and allocate memory
            :
[153]       decodedQueuePtr = decoderPtr->decodedQueuePtr;
            :
            // CIST BEGIN
[154]       TMqueue_push(decodedQueuePtr,...);
            // CIST END
            :
[159]       return;
[160] }

[1] TMqueue_push(queuePtr,...) {
[2]         pop = queuePtr->pop;
[3]         push = queuePtr->push;
[4]         capacity = queuePtr->capacity;
            :
[11]        if (newPush == pop) { // resize if needed
                    :
                    // malloc newElements array
                    :
[19]                elements = queuePtr->elements;
                    :
                    // populate newElements array
                    :
```

```
[36]                queuePtr->elements = newElements;
[37]                queuePtr->pop = newCapacity - 1;
[38]                queuePtr->capacity = newCapacity;
                    :
[42]        }       // end if resize
[43] elements = queuePtr->elements;
            :
[46] queuePtr->push = newPush;
            :
[48] return;
}
```

VITA

VITA

My research interests lie towards multi-core architectures and performance. I have worked on reducing hardware metastate and improving performance of hardware transactional memory (HTM) as part of my thesis work. My work was published in HPCA 2010 and ASPLOS 2013. I am also interested in other alternate interfaces of parallel programming as well as parallel programming abstractions to improve programability.

As part of my graduate research work I have also worked on on-chip networks. My first project reduced static buffer energy at low loads and was published at MICRO 2010. My current work aims at improving switch allocation performance and eliminating head-of-line blocking in the on-chip network. I have also developed and patented switch arbiter designs while interning at Oracle Labs. I am currently working on developing high performance network interface architecture in collaboration with AMD Research.

Aside from my work on HTMs and on-chip networks, I have worked on last level cache replacement and eviction policies with a view towards reducing write interference at the DRAM. I have also worked on architecture of web services, sensor networks, foreign language text recognition and digital watermarking of images.